The Use of Typed Lambda Calculus for Comprehension and Construction of Simulation Models in the Domain of Ecology

Michael Uschold

Ph.D.

University of Edinburgh

1990

I DECLARE THAT THIS THESIS HAS BEEN COMPOSED BY MYSELF AND THAT THE WORK DESCRIBED IN IT IS MY OWN:

Acknowledgements

First, I express deep appreciation for assistance provided by Alan Bundy, my supervisor. Time and again, he came through when it was required, in a way that no PhD student should ever expect. I thank him also for providing a highly conducive environment in which to do research.

I extend sincere thanks to Robert Muetzelfeldt, my second supervisor, who guided me through the complex domain of ecological modelling. A never-ending supply of challenging examples ensured that boredom never set in. The many conversations I had with Dave Robertson, and Mandy Haggith (members of the ECO research team) have been both enjoyable and indispensable.

On the non-technical side, I am eternally grateful to my special friend Karen McNish who supported me emotionally, domestically, and financially. Without her help, this would not have been possible.

I thank my parents for always providing excellent educational opportunities, and never pushing me in directions that suited them rather than me.

There are many others who have contributed to this effort. Special thanks go to Alan Smaill, Sean Mathews, and Colin Phillips for explaining what λ's and types are all about. Useful discussions about substructure and/or object-oriented representations were had with Geraint Wiggins, Robert Dale, Chris Mellish, Alex Laskarides, Colin Mathison, Chris Thornton, John Fraser, Robert Inder, and Ian Philby. Chats with Prem Devanbu clarified the relationship between this thesis and relevant work in the area of software engineering.

I thank collectively the Dreamers research group for constant support technically, and socially. Paul Brna helped me out of some serious LaTex binds. Geraint Wiggins and Frank van Harmlen provided excellent system support.

Desk space and computing facilities, were provided by the Artificial Intelligence Applications Institute.

I thank collectively other friends who helped me through the difficult times. Special mention goes to Alex Laskarides and Steve Senior. Finally, I extend sincere thanks to Mark Drummond whose friendship and technical awareness was a constant source of confidence and inspiration.

This research was supported by SERC grant GR/E/00730.

Abstract

We are concerned with two important issues in simulation modelling: *model comprehension* and *model construction.* Model comprehension is limited because many important choices taken during the modelling process are not documented. This makes it difficult for models to be modified or used by others. A key factor hindering model construction is the vast modelling search space which must be navigated. This is exacerbated by the fact that many modellers are unfamiliar with the terms and concepts catered for by current tools.

The root of both problems is the lack of facilities for representing or reasoning about domain concepts in current simulation technology. The basis for our achievements in both of these areas is the development of a language with two distinct levels; one for representing domain information, and the other for representing the simulation model. Equally importantly, we make formal connections between these two levels. The domain we are concerned with is ecological modelling.

This language, called *Elklogic*, is based on the typed lambda calculus. Important features include a rich type structure, the use of various higher order functions, and semantics. This enables complex expressions to be constructed from relatively few primitives. The meaning of each expression can be determined in terms of the domain, the simulation model, or the relationship between the two. We describe a novel representation for sets and substructure, and a variety of other general concepts that are especially useful in the ecological domain. We use the type structure in a novel way: for controlling the modelling search space, rather than a proof search space.

We facilitate model comprehension by representing modelling decisions that are embodied in the simulation model. We represent the simulation model separately from, but in terms of a domain model. The explicit links between the two models constitute the modelling decisions. The semantics of Elklogic enables English text to be generated to explain the simulation model in domain terms.

Inherent in this is a new approach to model construction which we have implemented in a computer program called ELK. Users build up a sequence of models, each being used to identify and constrain the important modelling decisions for the next one. The first model consists of general domain information (*e.g.* forestry). The second is a description of the particular situation of interest (*e.g.* some forest). Finally a simulation model of that situation is constructed. This approach enables users to communicate in familiar terms as well as significantly reducing the number of decisions that have to be made at any point. Constructing simulation models this way enables them to be self-documenting; this facilitates model comprehension.

Table of Contents

List of Figures

List of Tables

Part I

Analysis of Problem

Chapter 1

Introduction

1.1 Introduction

This thesis addresses two important issues in simulation modelling: *model comprehension* and *model construction.* By model comprehension we mean understanding the relationship between a simulation model and the real world system that it represents. Current software tools do not give adequate support for documenting important modelling decisions. This makes it difficult for models to be modified, or used by others. A key factor hindering model construction is the vast modelling search space which must be navigated. This is exacerbated by the fact that many modellers are unfamiliar with the terms and concepts catered for by current tools. The domain we are concerned with is ecological modelling.

Constructing simulation models is an important example of the more general *formalisation problem.* The question is: "How can a person convert their[1] informal and possibly vague understanding of a concept or problem into a formal specification?". To gain insights on how to go about solving our particular formalisation problem, we consider what has been done more generally. We identify some important difficulties, and some techniques that have been used to overcome them. We apply them to the domain of ecological modelling. We then generalise our experience and propose a framework for solving formalisation problems independent of our specific domain.

---

[1] We use *they, their, them,* etc. both in the singular and plural (like the word *you*) to avoid specifying gender and such awkward constructions as *s/he, him(his)/her,* etc.

In its most general form, the formalisation problem is vast in scope encompassing virtually all forms of modelling. Difficult and important formalisation problems arise in many contexts; some general techniques have been identified [Polya, 1945]. In this thesis we wish to explore the extent to which computer-aided assistance may alleviate the formalisation problem. We shall usually refer to the person who is trying to formalise something as a 'user', (or in our context: 'the ecologist').

Much work has been done related to this problem in various contexts. Most of this is embodied in the trend toward higher and higher level programming and specification languages. These are in principle easier to use, and help to alleviate the formalisation problem, but no matter how high the level, programming is still programming. The difficulties may be reduced, but they do not seem to go away, rather they resurface in different forms. In this thesis, we have the following general objectives in mind:

1. to identify key issues and difficulties relevant to the formalisation problem in general,
2. to relate these to the phenomenon of difficulties not going away, but resurfacing in different forms,
3. to find out to what extent it is possible to stop the infinite regress of reappearing difficulties and produce useful systems which alleviate the formalisation problem,
4. to discover general techniques that may be used to solve the formalisation problem in a variety of contexts.

In order to meet these objectives, it was necessary to explore the formalisation problem in some specific context. We chose to build a computer assistant to help construct ecological models. The domain of ecological modelling has the following desirable characteristics:

- There is a real need for assistance: many ecologists do not build computer models due to inadequate mathematical and/or computing skills.
- It facilitates a gradual approach by starting simply and increasing the complexity. This is because first, the domain of ecology is rich and diverse, providing challenges at almost any level of difficulty; and second, it is usually possible to focus on narrow subdomains without excessive loss of generality.

- It is an example of the following two more general domains, thus increasing the likelihood of being able to apply any techniques developed more widely:
  - software engineering
  - computer simulation

In this chapter, we outline our achievements regarding the first two general objectives of this thesis. The main body of this thesis describes our achievements of the third objective in the context of ecological modelling. Specifically:

> *We explore the nature of the work required to alleviate the problems ecologists have when formalising ecological systems.*

This work is embodied in a computer program called ELK. This is one of a series of many systems that have been developed in the context of the 'ECO Project' at Edinburgh University. Details of the relationship between ELK and the overall project are given in § 8.2.1[2].

In pursuit of the fourth general objective, a major [more specific] objective of this thesis was:

> *to identify how goals may be used to assist in the formalisation process.*

We refer to this as our *goals objective.* We take it as given that goals will be useful, the question is *how.* Goals serve as a useful starting point for identifying a wide range of issues.

1.2 The Formalisation Problem

To formalise something means to express that 'thing' using a specific set of syntactic and semantic conventions. We normally refer to this set of syntactic and semantic constructs as a formal language, or a formalism. Formalisms take various forms, *e.g.* logics, programming languages, specification languages, knowledge representation languages. The set of legal commands and input sequences of any interactive system also constitutes a formal language. By this characterisation, the problem of formalisation includes potentially all issues in user interface design.

---

[2] The team includes Alan Bundy, Dave Robertson, and Mike Uschold from the Department of Artificial Intelligence and Robert Muetzelfeldt from the Department of Forestry and Natural Resources.

Depending on the nature of the activity, the process of formalisation may be viewed as programming, creating a specification, or simply using an interactive computer system. We shall consider two different cases:

1. The problem is very well thought out. Users' ideas are not vague at all, merely informal. A detailed requirements document (written in English) for a software project is an example of this type.
2. The problem is not well thought out. Users' ideas are vague as well as informal.

In the first case the problem is one of converting the problem formulation (either a written document or a detailed formulation in a person's head) into the constructs of the formal language. This may be a straightforward translation, at one extreme; or an impossible task of representing a problem in an unsuitable formalism at the other extreme.

In the second and harder case, there are two conceptually distinct steps to formalisation. First, the vagueness of the users's ideas about their problem must be reduced and/or eliminated. This step can be referred to as problem formulation. Second, the user must encode this formulation into the constructs provided by the formalism. Whether done in a person's head, or with computer assistance, this formulation should not be done independently from the encoding step. That is, the target formalism should be considered during the problem formulation. Indeed, it would be possible for a user to formulate the problem in a very detailed and otherwise adequate manner, but *in the wrong terms.*

1.2.1 Sources of Difficulty

The difficulty of the formalisation process varies greatly. At one extreme, it can be relatively straightforward. At the other, an individual may be faced with an impossible task of representing a problem in an unsuitable formal language, akin to fitting a round peg into a square hole. We identify two major issues which determine for a particular problem how difficult it will be to formalise it.

1. *Adequate Formalisms*

   (a) *Syntactic adequacy*: The language constructs may be difficult and/or awkward to use. First order logic is an example of this for many people.

The Unix command language is another example of a formal language with unfriendly syntax.

(b) *Semantic adequacy*: There are two aspects to this:

- *expressive power*: is the language capable (however difficult it might be) of representing the required information?
- *conceptual distance*: the degree to which the terms and concepts that the user is thinking in (or in which the problem is formulated) are similar to the semantic constructs of the language. When this distance is large, significant difficulties arise. In [Robertson et al, 1988b] the term 'correspondence' was used to denote the inverse of this concept.

2. *Choice Management*: Formalisation problems frequently have large search spaces. In interactive systems there are two types of choice. The first is a control decision about *what* to do next. The second is a method-selection decision about *how* to do something. If there are many of either type of decision and if it is important to make good choices (either in terms of efficiency or accurateness of representation) then this too can greatly complicate the formalisation process.

On the issue of syntax, it is important to note that an underlying formalism may have a quite ugly syntax, but by adequate 'sugar-coating' this becomes irrelevant. This is discussed in § 1.3. Note that whether the syntax is good or bad is independent from whether the semantics is adequate. Assume for the moment that the syntax of a language is very good. This implies nothing about the expressive power or conceptual distance. There are many examples of systems whose interfaces are very slick, giving the effect of a very nice syntax. However, depending on what the language is for, it may be either very awkward (*i.e.* with adequate expressive power but large conceptual distance) or not adequate for the task at hand (*i.e.* insufficient expressive power). For example, Stella [Lewis, 1986] is a system which is very easy to use to construct an important but specialised class of simulation models (system dynamics models). It is based on graphic manipulation of icons. If the user wishes to construct system dynamics models, the conceptual distance is very small. If on the other hand, the user wishes to build models which are *not* readily representable in the system dynamics methodology, then the con-

ceptual distance is very large. It may not even be possible to construct the model using the tool. All the time, the syntax is very good.

Conversely, consider the case when the syntax is unfriendly. This also has no impact on the size of the conceptual gap. The conceptual distance may be minimal in spite of the poor syntax, rendering the problem of formalisation doable, if annoying and inconvenient. If the conceptual distance is significant, then we have the worst of both worlds. Unix is a good example to illustrate these points. Its syntax is arguably unfriendly and hard to get used to. However, for those familiar with the facilities that system command languages provide, it is a relatively simple matter to match up their terms and concepts with those of the language; thus the semantics is adequate. For someone new to computing, not only is the syntax terrible, but there is a large conceptual distance. This is because they are not trained to think in the terms and concepts offered by operating systems. Contrast this with the slick Macintosh interface which is easy to use by complete novices both because the semantics and the syntax of the interface is well-designed keeping the conceptual distance low.

Next we consider the issue of making choices. Largely, it is the size and complexity of the language, together with the sorts and sizes of problems for which it is intended, that determines how many and what kind of choices there are. If there are a large number of constructs, then the user may often be faced with both control and method-selection choices. If there are fewer constructs, then choices will be minimal. The issue of syntactic convenience is also independent from that of making choices. How convenient it is to code things in a language really has nothing to do with whether there are choices to be made. However, there is a connection between making choices and conceptual distance. For instance, if the conceptual distance is high, that means that the user will have to mentally translate their own terms into those provided by the system. All other things being equal, it is likely that there would normally be more than one way to capture a particular concept. If the conceptual distance was low, then that concept would [by definition] map fairly directly onto some construct. Thus, the number of choices of how to do things would tend to be larger with a larger conceptual distance. This means that in overcoming the difficulty of too great a conceptual distance, we simultaneously assist in choice management.

Until now, we have been assuming that the target formalism is a given and thus ignoring the issue of designing a suitable formalism in the first place. In some cases, a major part of the overall formalisation problem is itself concerned with designing a language which is sufficiently expressive, and has minimal conceptual distance with respect to the intended users. This is true for the whole field of knowledge representation. The field of knowledge acquisition is concerned with using the formalisms that have been created by the knowledge representation workers. These two activities are highly related. Designing a suitable formalism constituted a major part of the work in this thesis.

1.3 Solution Approaches

1.3.1 Syntax

Solving problems of poor syntax is possible by inventing a sugar-coated version of the syntax, and translating into the original language. Often, the new syntax is embodied in a user-friendly interface. For example, the rule acquisition system in Mycin [Buchanan & Shortliffe, 1985] provided a pseudo natural-language interface; the natural-language rules were automatically translated into Lisp. More generally, structured editors constitute a whole class of tools which are designed to alleviate problems of unfriendly syntax. Depending on the degree of 'coating', the syntax visible to and manipulated by the user may consist largely of the input commands of the user interface. An important design constraint in the design and implementation of ELK was to ensure that this sugar-coating would be easy to provide. This greatly depends on the closeness of the semantic primitives of the underlying language constructs to the terms that the user is thinking in.

1.3.2 Semantics

If expressive power is low, a formalism will be suitable for only a limited range of problems. If conceptual distance is high, lots of extra work and possibly 'hacking' may be required to achieve things for which the formalism was not intended. Solving these problems of semantic inadequacy consists of one or more of:

- extending the current formalism
- choosing another formalism

- designing a new formalism

If the conceptual distance is small and the problem is primarily one of inadequate expressive power, the most sensible thing to do is to try extending the current formalism if possible. If the expressive power is seriously inadequate, then this is is going to be a lot of work and may involve designing a new formalism. In this case, it might be preferable to choose a more expressive formalism, whose semantic constructs are similar, if one is available.

If the expressive power is adequate, but the conceptual distance is large, we must find a way to bridge this gap. This is a very common situation. Just about any computer programming exercise using low-level languages gives rise to such a situation. 'Low-level' is of course only relative. Throughout the history of computing, there has been a tendency toward higher and higher level languages. Initially, these were all programming languages, but now there are also specification languages, including so called wide spectrum languages. Some specifications are runnable and thus are also programs, but many are not [London & M., 1986]. In our terminology 'higher and higher' simply means reducing the conceptual distance. So, adequate expressive power in conjunction with large conceptual distance corresponds to a huge field of research. This includes designing and using programming and specification languages and thus is relevant to the whole field of software engineering. Our concerns are more directly related to the automatic programming subfield, particularly in assisting in the acquisition of specifications. This is discussed briefly in § 1.4.3 and again in chapter 8.

In this current situation (*i.e.* expressive power is adequate; conceptual distance large), extending the formalism is likely to be a major undertaking. It could end up being a kind of wide-spectrum language which contains constructs at various semantic levels varying from near to far from the terms that the user thinks in. Choosing another formalism may be possible depending on what is available for the problem at hand. Designing a new formalism might entail designing a specification language that bridges the gap between the semantic primitives of the user and those of the original formalism. There are two possibilities here. The first is to build a high-level compiler and the old formalism is no longer needed. The second is to write a translator from the new language to the original one. From the user's perspective, the effect is the same, the semantic primitives offered by the system are closer to their own. Note the similarity between the approach here and

that for solving syntactic problems. In this case however, instead of introducing a simple variation of the *syntax*, a variation or possibly complete redesign of the language is required. Instead of a simple interpreter/translator for converting syntactic variants of the same concepts, a rather more complex interpretation step is required.

The problem of conceptual distance can arise for reasons other than that the language is at too low a level. It can also be the case that the high-level language has the 'wrong' primitives. In this case, there is no point in extending the language. A new one has to be chosen or designed. As an example, a rule-based shell which is well-suited for diagnostic problem solving is not appropriate for specifying ecological simulation programs.

The worst of all worlds is to have inadequate expressive power in conjunction with a large conceptual distance. With respect to the majority of currently available languages and tools, this is the situation that exists in the domain of ecological modelling. Exceptions include special purpose tools with limited functionality (*e.g.* Stella [Lewis, 1986]). The analysis in the previous 3 paragraphs applies also to this case with the problems exacerbated by the lack of expressive power. In other words, it is a major undertaking to solve the problem.

1.3.3 Choice Management

Recall that there are two kinds of choices: *what* to do next, and *how* to do it. For the purpose of managing these choices we identify three levels of assistance that can be provided.

1. identification of potential choices
2. pruning inappropriate choices
3. advising on the best choice

The first kind constitutes defining the search space and serves to alleviate the 'blank sheet of paper' syndrome. This will frequently be a syntactic exercise. Pruning is especially useful if there are many possible choices. Some pruning may be achieved by syntactic techniques (*e.g.* type checking); more generally it will require domain knowledge. Advising is useful whenever it matters what to do or how to do something (*i.e.* almost always). It entails acquisition and application of domain-specific heuristic knowledge that can rank a set of appropriate options.

1.3.3.1 What to Do?

The decisions about what to do are important insofar as the problem should be approached in an orderly rather than random manner. It may be important to perform certain tasks before others, especially if there is much interdependence between different parts of the formalism. Identification of potential choices of what to do next may be largely syntactically determined (*e.g.* choosing from any construct in a language). These choices exist implicitly. If there is likely to be a great number, it may not be feasible to have them automatically generated and presented to the user; browsers may be required. In a complex formalism, there may be high-level tasks that naturally arise in the context of the intended use of a formalism, especially if one is limited to a specific domain. Then, the system may be equipped with a set of these high-level tasks which may be presented to the user. There may be various levels of tasks that apply in different stages in the formalisation process. In this case, the mere identification of what to do would be useful. However, it could be a significant effort to identify such tasks *a priori*.

An automated assistant may be able to prune some of these potential choices as inappropriate if the system is equipped with appropriate knowledge. Then, by analysis of the current state of the formalisation process (*e.g.* partial specification) it may detect inconsistencies.

The most advanced form of assistance is in advising the user about what they probably *should* do next. This may take various forms. At one extreme, the system may take control and completely relieve the user of any choices about what to do next. Alternatively, the user may have complete control, but the system offers advice in the form of an agenda of tasks with indications of whether, when and why it is important to do things in some specific order. Depending on the nature of the task, equipping the system with the necessary knowledge to make intelligent control decisions could be straightforward or exceedingly difficult. There may be no existing body of knowledge which can be used to guide the control. A heuristic approach may well be required. In the ecological modelling domain, this kind of knowledge is largely unavailable.

1.3.3.2 How to do it?

Assisting in how to do things is frequently much more important to the formalisation process than the control decisions just discussed. Correspondingly, it is much harder to provide useful assistance; it requires a deeper analysis of the domain.

Identification of choices about how to do something may also be a simple syntactic matter. For example, if a user chooses to use a certain construct in the language, then there may well be just a few different ways to do so. On the other hand, if it is a high-level task that the user has chosen, then complex planning may be required to identify options about how to do achieve that task. As was the case for 'what' choices, pruning and advising requires analysis of the current partial specification in conjunction with domain knowledge.

1.3.4 Reappearing Difficulties

Any attempt to alleviate a formalisation problem by building a computer assistant is likely to give rise to a new formalisation problem. In effect, the old formalisation problem is reformulated into the new one. The solutions that we have outlined above may be viewed as a set of methods for reformulating the formalisation problem. The success of the exercise depends on the extent to which the new formalisation problem is easier to deal with than the original one. We now consider how these difficulties reappear.

Consider, the syntactic problem addressed by structured editors. Suppose the primary formalisation problem is to use Pascal to encode some problem. The structured editor assists in this process. However, its allowable commands and input sequences also constitute a formal language. So there is really a secondary formalisation problem which is how to use the program editor's facilities. This may alleviate the syntactic problems, but other more fundamental problems require additional methods as described in § 1.3.

Consider the problem of too much conceptual distance. This may be addressed by designing a higher level specification language. From the user's point of view, the original problem is reformulated to one of creating specifications in the new language. This may also give rise to a new set of syntactic problems, and new problems of conceptual distance. Another potentially significant new difficulty that arises is translating the new specification into the original target formalism.

If these can be solved, they are transparent to the user and this constitutes a real achievement. If not, then there is a tradeoff.

Finally, we consider the problem of managing choices. If there are a lot of choices, then new syntactic problems arise in the context of developing an interface. Many techniques are available here including graphics menus and browser. Again, from the user's point of view, there can be a real gain; the new problem is much easier than the old one. However from the system developer's point of view, reformulation may give rise to a large amount of work in identifying and applying appropriate knowledge to guide choice making.

1.3.5 Summary: Solution Approaches

We identified the following fundamental difficulties faced during any formalisation process:

1. syntactic adequacy
2. expressive power
3. conceptual distance
4. managing choices

The major techniques that we have identified for overcoming these difficulties are:

- intelligent and/or user-friendly interfaces
- choice and design of formal languages
- interpreters/translators
- search space identification and control

Difficulties with syntax and conceptual distance may be alleviated by designing new languages in conjunction with interpreters or translators into a target formalism. In the case of syntax this usually is manifest in a user-friendly interface (*e.g.* structured editors). In the case of conceptual distance, it may or may not result in an interface. However the problems of language (re)design are likely to be much more significant. The idea is to have formalisms whose semantic constructs are closely matched to the terms in which the intended users think. The problem of interpretation/translation is also much more significant. In some cases, this proves virtually impossible (*e.g.* [London & M., 1986]) to achieve satisfactorily. Inadequate expressive power is handled by either extending the language, or choosing another.

There are three levels of assistance that may be provided for managing choices. Identifying the potential choices is frequently possible through syntactic analysis, although it may also involve considerable domain analysis. Pruning inappropriate choices may be done by a combination of syntactic analysis (especially type-checking) in conjunction with analysis of the current state of the formalisation process. To achieve this, the language should have a rich type structure and be able to represent domain knowledge. Advising on how to make good choices is a knowledge intensive exercise likely to involve a heuristic aspect. Good choice management is best achieved in conjunction with an intelligent interface. The extent to which the interface may be deemed intelligent depends in a large part on the extent to which the choice management involves pruning and advising as opposed to merely identification.

Because of the need for domain knowledge, constructing powerful assistants may only be possible in narrow domains. No single formalism is likely to have constructs which closely match the semantic primitives in a large variety of domains. Thus it will be rare for an existing language to be used in its raw form. Rather, for each domain, new higher level formalisms (or extensions to existing ones) will be required in conjunction with knowledge of that domain.

The common phenomenon in the successful approaches to solving the formalisation problem is that the bulk of the original difficulties with the formalisation process are transferred from the users to the system developers. For example, the reduction of conceptual distance itself tends to reduce the number of choices about how to do certain things. These problems are generally significant, as in the problem of interpreting/compiling/running high-level specifications, or going though an extensive knowledge acquisition to find out how to manage choices.

1.4 Related Fields

The main purpose of this section is to identify the niche for this research.

The nature of this project has been one of identification and synthesis of existing techniques and methods for the purpose of building a system which exhibited a functionality never before achieved. Below, we make clear the extent to which we can or have made contributions, or borrowed ideas from various related fields. By using and combining a variety of techniques in some novel ways we have made

notable contributions in a number of fields. The order in which we present these roughly corresponds to similarity of research goals and/or degree of direct relevance. In chapter 8 we explore in some detail similarities and differences between our methods and those used in these fields.

1.4.1 Ecological Modelling

The field of ecological modelling is most directly relevant; it is described in chapter 2. The contributions to this field are significant. In general terms, we have explored in depth how artificial intelligence techniques may be applied to assist in the comprehension and construction of ecological simulation models. By contrast, most existing work in this area has tackled different problems [Thomson & Taylor, 1990], or is speculative in nature [Loehle, 1987]. In specific terms:

- By designing and using suitable representations, we have mechanisms which facilitate automatic documentation of many of the modelling decisions. (Developed fully in this thesis)
- By developing suitable representations we have reduced the conceptual distance in this formalisation problem. In conjunction with suitable dialogue techniques we constructed systems which enable models to be constructed in ecological terminology, (not in programming terms). (See [Uschold et al, 1986, Robertson et al, 1988b])
- We have developed a specialised formalism based on typed lambda calculus for representing a wide range of ecological information. This representation is very general and may be put to a variety of uses. (See [Bundy & Uschold, 1989])
- Our methods facilitate considerable potential in reuse of a variety of ecological and modelling information/techniques for a variety of ecological models. [Muetzelfeldt et al, 1989]

1.4.2 Artificial Intelligence and Simulation

Many of the techniques that we have developed in the area of ecological modelling are fairly general; thus there is reason to be optimistic that these may be more widely applied in the domain of simulation, especially continuous simulation. Where AI and simulation overlap, the majority of work seems to be in the area of

discrete event simulation (*e.g.* [Fox, 1986]). We have explored the rather different area of continuous simulation. We are addressing two major issues:

- model comprehension
- model construction

To facilitate model comprehension, we aim to ensure that models are adequately documented so they may be readily used, modified, and extended. Documenting models means giving an account of the model in domain terms; *i.e.* the model assumptions should be explicit and examinable.

With respect to the latter, we are concerned with building an intelligent model-building environment usable by domain experts with minimal maths, modelling and computing skills. Various forms of intelligent guidance include managing choices, and ensuring consistency. Others are discussed in chapter 4.

Other major areas in the nascent field of 'artificial intelligence and simulation' that we are *not* addressing are listed below. The first two relate to how artificial intelligence can be useful in simulation; the latter point works the other way around.

- planning simulation experiments to achieve certain outcomes
- analysis and interpretation of model output
- use of simulation models to provide 'deeper' knowledge to enhance existing knowledge based systems

1.4.3 Software Engineering

There is considerable work being done in various subfields in software engineering which is related to our work. These are:

- Software comprehension
- Domain modelling
- Software reuse
- Requirements capture

Research in software comprehension [Letovsky, 1986], domain modelling [Biggerstaff, 1989], and software reuse [Lubars & Harandi, 1988] is relevant but has somewhat different specific goals than ours. There is much work related to semi-automatically building domain models of existing large software systems in a process of reverse engineering [Devanbu et al, 1990]. In this context, domain modelling

is seen as a means of achieving the goal of software comprehension. We approach the problem from the other direction. We use domain models to assist in the construction of simulation programs, but we share the ultimate aim of having well documented software. In ELK, an explicit account of [much of] the simulation model in ecological terms as well a record of [some of] the key modelling decisions is kept. A rich representation of the domain knowledge facilitates the ability of non-programmers to interact with the system in familiar terminology to describe particular ecological systems to be modelled. It is used to guide and constrain the requirements capture and model specification phases.

Reuse is an important goal in software engineering. See [Lubars & Harandi, 1988] for recent work in applying knowledge based approaches to software reuse. Although not a primary goal of our research, this has certainly been an important theme. In our work, reuse is evident in a number of ways. This is discussed in § 4.5.4.

Our work is similar in overall goals to the work in automating the requirements capture phase. For some time now, it has been realised that no matter how high the level of programming or specification language, the job of creating specifications or programs is still very difficult. There is a growing number of workers concerned with the problem of developing specifications [Fickas, 1987; Reubenstein & R., 1989]. The research strategy adopted by many of these workers is in the same spirit as ours. That is, they are working in specific domains with the aim of identifying the difficult problems in the hopes of later generalising them to other domains. There are also workers in automatic programming who, although not addressing the issue of creating specifications, they are, like us, concerned with developing non-trivial domain-specific applications programs [Barstow et al, 1982; Neighbors, 1986].

1.4.4 Human-Computer Interface

Although, our primary interest is not in the development of user-friendly interfaces per se, our work is inextricably linked with this field. First, we see intelligent and/or user-friendly interfaces as a vehicle for solving problems of formalisation. This role of interfaces is described in § 1.3. Second, any interface to any computer system constitutes a kind of formal language, and thus the whole field of human computer interfaces can be said to be addressing a particular kind of formalisa-

tion problem. Accordingly, our general approaches towards solving formalisation problems should apply.

For example, using the terms introduced in § 1.2.1 we can define a 'user-friendly' system to be one for which the problems of formalisation go away. This happens if/when:

- the syntax of the language is suitable (*i.e.* the user commands and options are easy to use)
- the conceptual distance between the user and the semantic constructs of the interface language becomes negligible,
- the expressive power of this language is adequate (*i.e.* computer system enables the user to accomplish what they require)
- the choices are manageable (*i.e.* either there are not very many options, or it is easy to choose between them).

This ignores a great many specific issues in user interface design many of which we are not concerned with (*e.g.* psychological studies, key-stroke analysis). [Robertson et al, 1988b] presents an extensive analysis of the relationship between the formalisms used and the user interface.

A highly relevant work which influenced this research is [O'Keefe, 1985]. This was an extensive exploration into the problem of building an intelligent assistant for statistical analysis (ASA). We compare ELK with ASA in § 8.5.

1.4.5 Knowledge Representation and Acquisition

The formalisation problem is at the heart of the related fields of knowledge representation and acquisition. Knowledge representation and acquisition are relevant to our project in two distinct senses. First, one can view ELK itself as a knowledge acquisition system. It is used to acquire ecological knowledge, which guides and constrains possible descriptions of ecological systems that are acquired. ELK also acquires simulation models of ecological systems. So, in this first sense, the users have to represent knowledge in terms that the system 'understands'.

Second, in developing a computer assistant for ecological modelling, we found it necessary to acquire much ecological and modelling knowledge as well as design suitable formalisms for each. Thus, in our effort to solve the formalisation problems of ecologists, we have been forced to embark on a formalisation process of our

own. However, ours was a manual exercise for which no computer assistance was available.

Much of the research in the fields of knowledge representation and acquisition can be viewed in terms of our analysis of the formalisation problem. For example, the knowledge representation workers are forever designing new languages to both increase expressive power and to provide the 'right' semantic primitives so that the intended users of the formalisms can represent what they require with minimal conceptual distance. The kinds of formalism available define choices at one level. One of the aims of knowledge acquisition research is to identify criteria and guidelines which can be used to make choices about which representations are good for certain sorts of problems. In the very long term, this information may find itself in computer assistants, but at this time the majority of the core problems faced by knowledge acquisition workers are being dealt with manually. They result in methodologies and guidelines, sometimes partially embedded in computer systems (*e.g.* KADS [Hayward, 1988] and KREME [Abrett & Burstein, 1987]). Of the automated techniques that are available, many are useful only after a significant amount of knowledge identification and acquisition has already been done. For example attributes, values, and decision classes must be identified before an induction system is usable.

In struggling with representation issues, we were able to make some contributions related to the knowledge representation field. Our development and use of a representation based on the typed lambda calculus is described in [Bundy & Uschold, 1989]. We have made no specific contributions to the knowledge acquisition subfield per se, but some of our techniques are similar to those in KREME.

1.5 Thesis Outline

PART I: ANALYSIS OF PROBLEM

1.5.1 Introduction

We introduce the formalisation problem and note that although many techniques and approaches have been used in many contexts, the inherent difficulties do not seem to go away; rather they resurface in a different form. We seek a general methodology to stop this infinite regress of reappearing difficulties whereby useful

computer assistants can be produced for an important class of formalisation problems. We propose to build a computer assistant for creating ecological simulation models and generalise the results of this exercise.

In the simulation modelling domain, two key issues arise: *model comprehension* and *model construction*; these are the focus for the thesis.

1.5.2 Ecological Modelling

We introduce ecology and ecological modelling. We note major model types and modelling paradigms. We are concerned with differential and difference equations models. We motivate the need for a computer assistant by noting some current problems with ecological modelling. The chief issues we are concerned with are (1) model comprehension and (2) the process of constructing models. We describe a simple model of part of the Serengeti ecological system and use it to illustrate the difficulties with current simulation support tools. We describe the range of ecological information that we must represent. We introduce the kernel of modelling concepts that are required to represent computer models of ecological systems.

1.5.3 Ecological Modelling Goals

We explore the nature and potential uses of ecological modelling goals. We begin with a brief discussion of the modelling process in general, and the role of goals. We present a classification of goal types and uses resulting from a survey of selected texts and papers in the ecological modelling literature. We outline the requirements for representing some of these goals. We conclude that the major role of goals initially is to identify what the important things are in the ecological system and corresponding simulation model. We show how the goal classification might be used as the basis for a control strategy for a model acquisition system.

1.5.4 Design Considerations

We give the design rationale for ELK. We describe how the general difficulties in the formalisation process manifest themselves in the ecological modelling domain. We note various other important difficulties that arise in this domain. Each major difficulty gives rise to a major design requirement (*i.e.* to overcome it). We

examine each of these, describe further requirements that they give rise to, and ultimately suggest appropriate techniques to meet these requirements.

A key aspect of the design is a three layer *knowledge ontology* which distinguishes general/ecological knowledge, ecological system descriptions, and modelling information. This, combined with a general formalism with relatively few primitives and many ways to combine them facilitates various important benefits for the model acquisition system. We state two key hypotheses with respect to this ontology:

1. *ontology completeness*: all the information relevant to ecological modelling can fit into this ontology
2. *ontology usefulness*: the distinctions that the ontology embodies are useful.

PART II: A SOLUTION

1.5.5 ElkLogic

We describe ElkLogic, the theory underlying the representations used in ELK. It is based on the typed lambda calculus. Of particular importance are (a) the rich type structure, (b) the representation for sets and substructure and (c) the use of higher-order functions (d) semantics. Together these features facilitate (a) keeping the number of primitives small, (b) reduced search, and (c) explicit representation of the ecological meaning of model variables.

The general/ecological knowledge base includes taxonomies of sorts of ecological entities and their parts, the attributes that these sorts of entities have, and the processes that these entities participate in. The description of the ecological system consists of creating specific entities, specifying their substructure, and the processes that they participate in. The model information consists of (a) definitions of variables and parameters corresponding to attributes and effects of processes and (b) equations for computing values of variables.

1.5.6 Elk: Representation

We describe the implementation of the representation formalism, ElkLogic. We give the details of an additional level in the knowledge ontology which serves to bridge the conceptual distance as well as reduce the necessity for menial tasks

for the user. This includes specifying goals, what the user is interested in, and defaults.

We investigate the ontology completeness hypothesis by placing every construct in the representation in our ontology. We illustrate every important construct in the language using examples from the Serengeti model presented in chapter 2.

We begin to investigate the ontology usefulness hypothesis by explaining how it facilitates meeting the major design requirements regarding expressive power, conceptual distance, and model comprehension.

1.5.7 Elk: Elicitation

We describe how ELK may be used to elicit formal descriptions of ecological knowledge, systems, and models. This entails giving a general overview of the interface, the facilities that it provides, and some general guidelines on how it is intended to be used. We describe in turn three phases in the modelling process. During one phase, a general/ecological knowledge base is created. Another phase consists of describing the particular ecological system of interest. The last phase consists of specifying the simulation model. Allowing users first to say what they are interested in before specifying the runnable model is a key aspect of ELK.

We describe various aspects of the process of gradual elaboration, which is used to break down the complex task of constructing an ecological model into simpler subtasks. The information given by the user initially is fairly general and underspecified. Nevertheless, it provides hooks for the acquisition of more and more detailed information ultimately resulting in a detailed formal description of the ecological system, the goals, and the model. The user has initiative most of the time ensuring flexibility in terms of what they do and when.

We reconsider each of the major requirements laid down in the design rationale in chapter 4. We explain which techniques are important in addressing which requirements. Of particular importance is the issue of where and how choices arise in the elicitation process. The rich type system plays a significant role in defining the search space. The distinctions embodied in the knowledge ontology give direct support to most of the major requirements. This supports the ontology usefulness hypothesis.

PART III: DISCUSSION

1.5.8 Related Work

We consider the related subfields of ecological modelling, artificial intelligence and simulation, software engineering, intelligent user interfaces, and knowledge representation. We compare the strengths and weaknesses of our techniques with other systems and approaches. We also indicate important differences in emphasis with respect to other research projects. We summarise the main contributions of the thesis with respect to each area.

1.5.9 Summary and Conclusion

We give a chapter by chapter summary of the thesis, noting the central problems addressed and the main techniques used. The choice of representation is the key to our progress towards solving *both* of our fundamental problems: model comprehension and model construction. Though our progress has been significant; we do *not* claim that we have fully solved the problems.

The overall approach, and most of the specific techniques that we develop are independent from the ecological domain, and apply more generally to the field of simulation modelling. We further generalise our experience and propose a methodology for developing computer-aided assistants to formalisation problems in general. Finally, we outline how this research may continue.

1.6 Conclusion

We have introduced the general problem of formalisation. We noted some general difficulties that arise. We outlined approaches to overcoming them in the context of building a computer assistant. We conclude that the enterprise of constructing such an assistant consists of reformulating the original formalisation problem into a new one. In the new formulation, the same difficulties may arise in principle, but in a new form which should be easier to cope with.

We propose to explore the formalisation problem and test our suggested solution approaches in the context of building computer models of ecological systems. This thesis describes our results. In the next three chapters, we motivate and describe the design of ELK, a computer assistant for ecological modelling.

Chapter 2

Ecological Modelling

2.1 Introduction

This chapter describes the domain of ecological modelling. We begin with a description of what ecology and ecological modelling is and why it is useful. We classify some of the major kinds of models, and modelling paradigms and note which ones we are concerned with. This is followed by comparison of ecological modelling in particular with formalisation problems in general. We note how the general difficulties described in § 1.2.1 arise in this context and how the solution approaches described in § 1.3 might apply. We outline some major problems with the current state of the art in tools which support construction and use of ecological models. These serve as important foci for this research. We present a simple but non-trivial ecological model which serves as the running example throughout this thesis.

We distinguish two kinds of information which we refer to as the *ecological level* and the *simulation modelling level.* We consider what kinds of things need to be represented for each. In doing so, we address the issue of expressive power which was cited in chapter 1 as a major source of difficulty in the general formalisation problem.

Terminology

Appendix A contains a glossary of key technical terms used in this thesis. These are introduced and defined in the text as required. First, we do not make a technical distinction between the terms 'information' and 'knowledge'. We use

whichever term subjectively 'sounds better' in a given context. Next we clarify our use of the term 'model'.

conceptual model: a representation of one or more concepts and/or processes which may or may not be manifest in a computer program (*e.g.* an animal taxonomy; a predator-prey model).

computer model: the embodiment of one or more conceptual models in a formal representation intended for computer processing. We distinguish two kinds of computer models:

static model: a computer model which 'merely' represents static information; a data structure. (*e.g.* a taxonomy of species)

simulation model: a computer model which is a program that simulates a dynamic system that the embodied conceptual models collectively represent (*e.g.* of an ecological system). A simulation model usually contains static models.

We distinguish also between a modelling framework (or paradigm), a modelling formalism, and a model. This is analogous to the distinction between a programming paradigm, a programming language and a program. The framework is an ontology on which a formalism for defining models is based. The formalism is used to create specific models. For example, our conceptual modelling framework is based on five fundamental concepts required to characterise the ecological domain. These are processes (*e.g.* predation), entities (*e.g.* lions), substructure (*e.g.* male/female lions), attributes (*e.g.* weight), and values (*e.g.* 43 kg). We refer to it as the PESAV framework. Our simulation modelling paradigm is based on differential and/or difference equations. In chapter 5 we define a wide-spectrum language which may be used to represent both static and simulation models of ecological systems. In the context of ecological modelling, the term 'static model' is synonymous with 'description of the ecological system to be simulated'.

Finally, we note introduce some terminological conventions. We use the following conventions for abbreviating frequently used long-winded terms like 'ecological simulation model'. Unless otherwise specified or clear from context:

- 'model' is short for 'simulation model'
- 'X' is short for 'ecological X'

Thus both 'model' and 'ecological model' will usually mean ecological simulation model. The latter convention is more general and will apply to a variety of terms including 'entity', 'process' and others.

2.2 Ecology and Ecological Modelling

2.2.1 Ecology

Ecology is the study of plants and animals in their environment[1]. Ecologists are concerned with understanding the plethora of interactions that take place in ecological systems. Such understanding enables them better to predict the outcome of such interactions, in terms of how many of which organisms are where. The ability to predict, combined with mankind's ability to cause change gives the potential for safe management of our environment. For example, overuse of harmful pesticides has prompted searching for natural alternatives; also there is much concern about the greenhouse effect.

Ecological systems are highly complex, involving many interactions between organisms and various environmental factors. Furthermore, they may be viewed at many different levels of detail. At one extreme, an ecologist might be interested in studying the competition for light between individual needles on a pine branch. At the other extreme, the interest may be in the effect of the rain forests on the carbon dioxide levels in the atmosphere. In between these extremes, ecologists study interactions between individual plants and animals as well as populations, communities, and whole ecosystems. For these reasons, the study of ecology is extremely challenging.

2.2.2 Ecological Modelling

Modelling is an important tool used by ecologists. Models are a useful medium for sharing knowledge. They formally represent an understanding of how an ecological system operates which can be evaluated by others. There are many kinds of

---

[1] Most of the material for sections 2.2.1, 2.2.2, and 2.2.3 is drawn from [Robertson et al, 1991].

models and corresponding paradigms. [Robertson et al, 1991] notes three major kinds of models: *statistical models*, *theoretical/analytical models*, and *simulation models*[2]. The first kind are used to summarise large amounts of real data but tend to ignore underlying mechanisms. The second kind tend to be fairly abstract; mathematical tractability often is often given higher priority than ecological sense. Although directly concerned with mechanisms, gross assumptions are made to ensure tractability and real data tends not to be used. Because these approaches ignore real data or underlying mechanisms, or make gross assumptions their results lack obvious meaning. Such models are certainly not usable for prediction or management.

Simulation models are solved by iteration over a set of time steps. The model consists of a number of variables and parameters which describe the state of the system being modelled. All the variables must be initialised, and the parameters given values. The simulation consists of recomputing the values of all the variables at each time step and outputting the results as needed. In general, the values of the variables at a time step depend on their values at the previous time step.

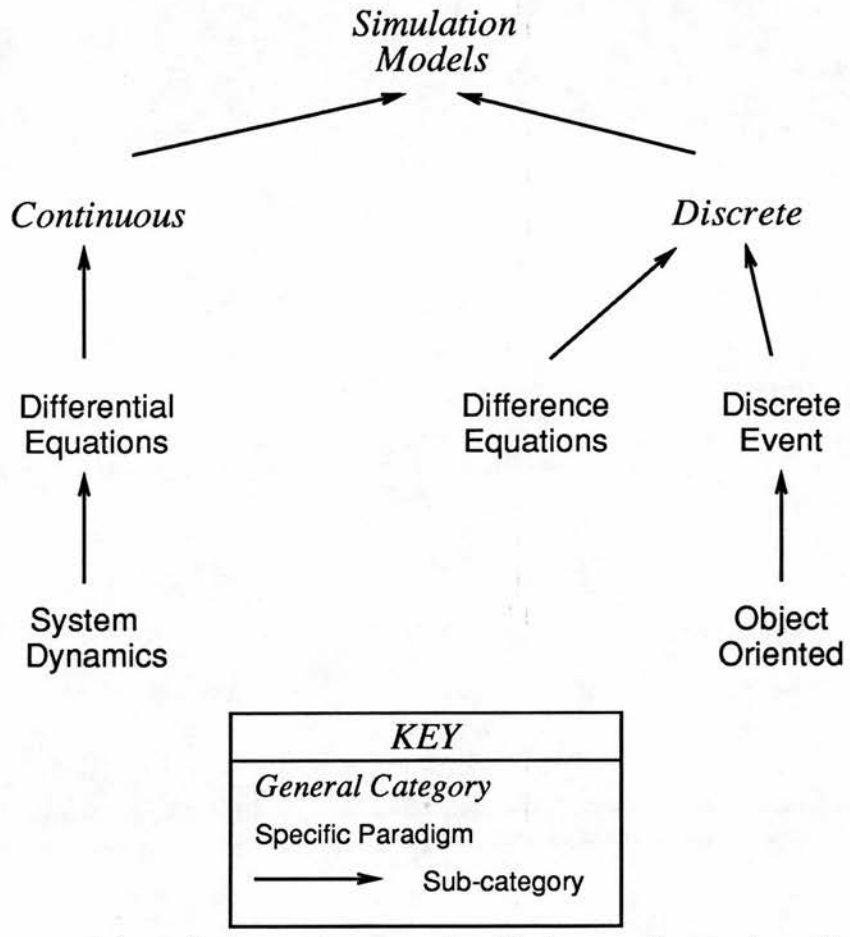
It is possible to build simulation models which represent underlying mechanisms, make realistic assumptions, and use real data for calibration. Such models, in their limited domain of application, have the potential for being both meaningful and accurate and thus much more useful than either of the first two kinds. In this thesis, we are concerned only with simulation models.

2.2.3 Simulation Modelling Paradigms

Simulation models, and corresponding paradigms, are frequently classified according to how time is handled (see figure 2–1). A fundamental distinction is whether time is continuous or discrete. Continuous models usually consist of a set of differential equations. These are appropriate for continuous real world phenomena (*e.g.* flow of water). However, they are also used to approximate real world events which are discrete (*e.g.* births and deaths, radioactive decay). Usually, differential equations models cannot be solved analytically. Various numerical

---

[2] [Robertson et al, 1991] uses the term 'numerical simulation model' here. Although numerical computation is by far the norm, we allow for modelling changing colours and other non-numerical things.



Simulation models and approaches are classified according to how they represent time. Note that these categories are not all mutually exclusive. For example, differential equations are frequently approximated by difference equations.

Figure 2–1: Simulation Modelling Paradigms

techniques are available including for example, Newton's method, and the Runge-Kutta technique. The former is a particularly simple technique used by ELK.

System dynamics models are a particularly useful subset of differential equations models. They are most appropriate when the system can be conceived of as a set of compartments and flows. Many ecological systems can be thought of in this way. The fact that ecological systems naturally map onto the system dynamic framework means that the conceptual distance is small, greatly simplifying the formalisation process.

> "...it is possible for an ecologist to specify complex and realistic [system dynamics] models without entering a single equation ([Robertson et al, 1991], chapter 2)"

Consequently, this technique has been widely used; *e.g.* to model the transfer of biomass energy, nutrients, etc. We shall have frequent occasion to refer to system dynamics models in this thesis. Readers unfamiliar with this paradigm are referred to appendix B.

Difference equations may also be used directly to represent ecological systems which are not conceived as being continuous. An example of where difference equations are appropriate is reproduction, which for many species takes place annually. Running such models can be identical to running a differential equations model if Newton's method is used with a suitably coarse time step. This method entails turning the differential equations into difference equations.

Discrete event simulation is similar to difference equations; the major difference is that the time step is not evenly spaced. Simulation time moves forward when the next interesting event takes place. This is often used as a substitute for continuous simulation where numerical methods are error-prone. For example, a well known problem is in choosing the time step. Although extensively used in other domains, (*e.g.* job-shop scheduling) this technique is rarely used in ecology (see [Saarenmaa et al, 1988; Folse et al, 1989] for exceptions).

Another very widely used technique outside the ecological domain is object-oriented modelling. This technique is appropriate when the world can be readily viewed as consisting of objects, each having a set of attributes, and/or properties that define and describe them. For example, sheep may be defined as a type of mammal with specific properties. Individual sheep may be described by attributes like weight, height, etc. Object-oriented representations are especially well suited

for taxonomic classifications whereby more specific types of objects inherit properties from more general ones (*e.g.* sheep inherit properties from mammals). Interactions are modelled by passing messages back and forth between objects. For example an object representing the class of sheep might be sent a message to create a new instance. This could be used to model reproduction. Object-oriented modelling has not been widely used in ecological domains. Exceptions include modelling the creation and destruction of individuals for which the approach is well suited.

One of the main advantages of object-oriented approaches is that people often find it quite natural. Thus, it has the same important property that system dynamics modelling has with respect to to ecological domains: it is very easy to conceive of the real world situation in terms of the modelling framework. In other words, conceptual distance is small. It is surprising that object-oriented techniques are not more widely used for ecological modelling, for exactly this reason.

Finally, note that many object-oriented tools are fairly general purpose; *e.g.* they are convenient for implementing discrete-event models.

Choosing a Paradigm

The appropriate modelling approach depends on a variety of things. Most importantly, it depends on the reasons for constructing the model including its intended uses. All this comes under the umbrella of modelling goals. Models are used as a means of testing an ecologist's understanding of ecological systems. This in turn often leads to increased understanding. Prediction and management are other important reasons for building models. Chapter 3 considers the nature of ecological modelling goals and how they may be used. Other important factors relevant to determining the appropriate kind of model are:

- the level of detail required
  e.g. individuals or populations
- the amount that one knows about the system
  e.g. available field data
- availability of and familiarity with modelling tools and techniques.

In this research, we make no attempt to cater for all of these paradigms. We limit ourselves to differential and difference equations models. Because ELK uses Newton's method, which turns differential equations into difference equations it

(*de facto*) does not distinguish between difference and differential equations. If other integration techniques were used (*e.g.* Runge Kutta), then ELK could make this distinction. We are aware of no word which encapsulates both differential and difference equations models. Instead of inventing one, we state here that *for the remainder of this thesis, when we refer to differential equations we also mean difference equations.*

The representation developed in this thesis has its roots in both system dynamics, and object-oriented representations. We are careful here to distinguish object-oriented *representations* from object-oriented *programming* or modelling. The former are characterised by hierarchies of object types or classes (*e.g.* animal taxonomies) where each objects inherit properties from their classes and superclasses. It is roughly true that one obtains object-oriented programming by adding message-passing and code-sharing to object-oriented representations. We do not cater for object-oriented modelling which always uses object-oriented programming.

A major theme in this thesis is to minimise conceptual distance. We extend the system dynamics framework but retain the flow analogy as an option so that when it is appropriate conceptual distance is small. The object-oriented elements in our representation are used for exactly the same reason: to reduce conceptual distance.

2.2.4 Ecological Modelling as a Formalisation Problem

The formalisation problem that we are concerned with is how to construct simulation models of ecological systems. In § 1.2 we identified four potential sources of difficulty in any formalisation process (syntax, expressive power, conceptual distance, and choices). We discuss how these are manifest in the domain of ecological modelling. The first three relate to modelling languages of which there many. We consider two extremes. First, there are general purpose languages; *e.g.* Fortran is heavily used for ecological modelling. There are also a variety of more specialised languages which are appropriate for a smaller class of problems. For example system dynamics is a framework appropriate for problems that can be conceived as consisting of tanks and flows (see § 2.2.3 and appendix B).

syntax/interface: Are the language constructs easy to use and read? Generally no, for Fortran, especially for computer-naive ecologists. Syntax for spe-

cialised languages can be equally awkward, but are often treated with effective 'sugar-coating' in the form of a user interface (*e.g.* Stella [Lewis, 1986]).

expressive power: Does the simulation language have the capacity to represent a sufficiently wide range of models of the required kind? General languages do, more specialised ones can have a limited range of applicability.

conceptual distance: Does the language provide primitives which map naturally onto the way the ecologist is thinking about the problem? The answer is no for a general language like Fortran, but yes for system dynamics, but *only* for the right kind of problem.

choice management: In the ecological domain, the problem of choices is extreme. There are so many things to do in so many different ways that even experienced modellers can be overwhelmed with a complex problem.

Note that the first three difficulties are substantially overcome when using a specialised tool like Stella in its domain of applicability. These tools also help control choices by limiting the things possible to do to the narrow area of applicability. We require a more general tool which can express a wider range of models. This means that the major difficulties above rear their ugly heads. Extending the expressive power means the conceptual distance and the choices come back. These only went away by virtue of limiting the range of applicability. It also potentially exacerbates any syntactic problems, although this is a lesser concern.

To overcome these difficulties, we applied the four general solution approaches outlined in § 1.3. By doing so, we reformulated the formalisation problem that ecologists normally face into a [we claim] much easier one. Specifically,

- We built a computer assistant.
- We designed and implemented a formal language for representing the required information.
- We implemented a variety of translation mechanisms, and an interpreter for running the simulation models specified at a high level.
- We identified a significant portion of the modelling search space, and used consistency checking and heuristics to prune and advise on making good choices.

The issue of conceptual distance impacts heavily on the requirements for designing both the formalisms and the mechanisms for providing assistance. Small

conceptual distance will likely facilitate designing a more effective assistant (*i.e.* easy to understand and use); but may make it harder to design the intermediate (or wide-spectrum) language and corresponding mechanisms for translation into the target formalism. Conversely, large conceptual distance makes it harder to design an effective interface but simplifies the language design.

The bulk of our efforts have centered around the identification and categorisation of the required knowledge and corresponding design of a suitable formalism for representing and reasoning about this knowledge. We not only had to be able to represent a wide range of simulation models, but we also had to represent a wide range of ecological knowledge. Furthermore, we required mechanisms for bridging the two. Forming this bridge is the essence of ecological modelling.

Our approach was to construct a kind of wide-spectrum language based on the typed lambda calculus which represents both kinds of information as well as the bridging mechanisms. This simultaneously addresses both major issues in this thesis. First, it enables the simulation model to be explained in terms of the ecological system being represented (model comprehension). Second, it reduces conceptual distance which makes model construction much easier. Many of the language constructs are cast directly in terms that the user thinks in, thus ensuring that there would be minimal difficulty in providing easy to use interface primitives. Another primary factor was controlling the modelling search space. The use of a language with a rich type structure was instrumental for this.

2.2.5 Problems with Current Technology

Ideally, it should be possible for any ecologist to formalise an ecological system using a modelling framework which allows it to be easily accessed, analysed, assessed, modified and/or extended by other researchers. However, most current 'representations' of ecological models are unsuitable for this. For example a Fortran program, while highly formal and unambiguous, is usually inadequate for the purpose of understanding the underlying model; *i.e.* the nature and extent to which the real ecological system has been idealised. Such information is found in journal articles describing models, however this is highly informal and often ambiguous. They are usually inadequate for the purpose of constructing or running the model as a computer simulation. Here we elaborate on the two key

issues mentioned in § 1.4.2 which provide the main focus for this research: *model comprehension* and *model construction.*

To have achieved model comprehension we mean that the model can be examined and understood because an account of the model in domain terms is available. The important facts about the system being modelled are available as well as the important modelling decisions related to these facts. No simulation technology that we are aware of addresses this issue.

Lack of model comprehension is the source of many problems. First, it makes it difficult or impossible to analyse a model. Second, it makes it difficult or impossible for it to be safely used by others. New users must know the important assumptions on which a model is based to know when and whether it is suitable for their problems either as a stand-alone model, or as part of a larger model. It is essential to know a model's limitations when interpreting the results. Third, lack of model comprehension make it difficult to modify or extend the model. It should be possible to experiment with models by making various changes in the model assumptions which result in automatic updating of the underlying implementation. Because such assumptions are never recorded, this is impossible.

The key problem with respect to model construction is that many ecologists do not have the mathematical, modelling, or programming skills needed to construct ecological models (*i.e.* the conceptual distance is too great).

Note that these problems are not at all specific to the ecological domain but apply generally to computer simulation. In light of this, the key objective driving this research in the context of ecological modelling is to create a tool which:

- supports the construction of models which incorporate representations of important knowledge about the model (this addresses the model comprehension issue).
- enables ecologists, whether expert modellers and programmers or not, to construct models by communicating primarily in ecological and modelling terms (this addresses the model construction issue).

We have achieved the first objective, and made substantial progress on the second. We have developed a new approach to modelling which works in the intended manner, however it has not been tested by a significant number of ecologists.

In this thesis, we describe a tool which incorporates explicit representations which support model comprehension. It is necessarily couched in ecological and

modelling terms. It uses a formalism which can represent both highly informal knowledge about the ecological system, reasons for wanting a model in the first place, and a host of assumptions that underlie the model *as well as* the myriad of formal details required to run the model as a computer program. In facilitating model comprehension, we simultaneously go a long way towards reducing conceptual distance. This is the major factor in facilitating model construction.[3]

Crucially, the representation of the ecological facts and assumptions must be distinct from but explicitly related to the representation of the runnable model. This gives rise to two distinct levels of information:

ecological level: including general knowledge about the ecological domain as well as the description of the specific ecological system to be modelled.

simulation modelling level: including goals, modelling decisions and assumptions, as well as the specification of the runnable model (including equations, parameter values etc.).

We require mechanisms for bridging these two levels. The need for this is an immediate consequence of the need to facilitate model comprehension. The distinction is also useful in identifying the search space for a suitable model of an ecological system. This chapter deals only with representing ecological systems and models. Chapters 3 and 4 consider various bridging representations and mechanisms, and introduce further distinctions within each of these two levels of information. In the next section we introduce a simple model which serves as the running example in this thesis.

---

[3] Historically, it was the other way around. We set out to build a system that communicated in ecological and modelling terms and discovered that in doing so there were necessarily a lot of hooks present which could be used to facilitate model comprehension. This then became a focus of the research in its own right.

2.3 Example Model: Serengeti

In this section we present a typical ecological modelling exercise described in [Hilborn & Sinclair, 1984]. This example served as a major source of design requirements in terms of what kinds of ecological and modelling information needed to be represented. It will be used throughout this thesis to serve as a focal point for considering a wide range of issues. The features which make this example adequate for these purposes are:

- The Serengeti is an extremely rich and diverse system which may give rise to a wide range of possible models ranging from very simple to very complex.
- There is ample discussion of the ecological system being modelled (*i.e.* in [Hilborn & Sinclair, 1984]).
- There is ample discussion of the goals and objectives which motivated the modelling exercise.
- There is ample discussion of the modelling assumptions.

We begin with a brief overview of the Serengeti ecosystem [Sinclair & Norton-Griffiths, 1984] and of the particular aspects of the Serengeti that are to be modelled [Hilborn & Sinclair, 1984]. This is followed by a presentation of a complete yet simple model. In § 2.4, we discuss the kinds of modelling knowledge that are required. In § 2.5 we discuss the kinds of ecological knowledge that need to be represented. This prepares us for a detailed discussion of criteria for designing ELK. These criteria and an outline of the system design are presented in chapter 4. All of our concerns are ultimately related to the two primary issues of model comprehension and model construction (discussed in § 2.2).

2.3.1 Overview

The Serengeti lies between 1 and 3 degrees south of the equator in Acacia savannah woodland. The most general goal of the [ecological modelling] exercise is to understand the dynamics of the system. They are concerned with predicting the effects of a recent perturbation to the system, a climatic shift in the form of increased dry-season rainfall. There are two seasons, wet and dry, which occur from November through June and July through October, respectively. During the wet season, there is plenty of grass growth to provide food. In the dry season, the soil

dries out, and little grass growth occurs. In response, the wildebeest population migrates from the plains to the woodlands where there is more food available year round. The wildebeest do not stay in the woodlands year round because they are preyed upon by non-migratory lions and hyena in the area. Recently, the amount of dry-season rainfall has increased significantly from 150mm to 250mm. This has resulted in increased amounts of grass all year round. Furthermore, grass began to grow in the plains in the dry season when previously there was none. Dry season mortality of the wildebeest virtually ceased, resulting in a dramatic population increase.

The specific objectives of the modelling exercise given in [Hilborn & Sinclair, 1984] is to determine some of the particular effects this may have on various elements in the system. The authors' primary interest is in the wildebeest population itself. Will it reach equilibrium? How big will it get? Could a return to previous levels of much lower dry season rainfall lead to a catastrophic decline in population? Additionally, they want to know what if any effect the increase in wildebeest population may have on the alternate prey of the lions and hyenas. They list several possibilities, and wish to determine which of these is most accurate Also, they wish to identify what areas in the system are ill-understood thus pointing the way for new areas for field studies. They do not indicate a direct interest in the effect that the climatic shift has on the populations of the lions and hyenas.

2.3.2 A Simple Model

A simplified model was constructed for the purpose of determining the gross dynamics of the system. The model assumes that the predator and alternate prey populations are constant, and that wildebeest can be represented by a single age class. A simple predator-prey model is used which takes into account such things as rate of search, probability of catch, and density of prey populations. In some cases the parameters were observed directly; others were estimated or computed from observed data. The time scale used is annual. Rainfall amounts refer only to the dry season, and are constant throughout the simulation run.

The core of the model (excluding input/output) is coded in a simple Fortran program in figures 2–2 and 2–3. There are ten equations which we describe in turn. The letters **A**–**I** correspond to the comments identifying the equations in the Fortran program (figure 2–3).

A: Dry Season Grass Abundance It has been observed in the field that rainfall is by far the most significant factor in determining grass production in the dry season. This equation is derived by fitting a curve to existing data. It ignores all factors except rainfall.

```
GRS_WT = 200 + DRY_SSN_RAIN*2
```

B: Calf Survival Similarly, it has been observed that grass abundance in the dry season is the most significant factor in determining wildebeest calf survival. Again, the precise relationship is derived from field data. Other factors are ignored.

```
SPR_CF_SURV = (GRS_WT * .05) / (75 + GRS_WT)
```

C: Gross Wildebeest Reproduction The number of new calves born each year is the product of the number of wildebeest and the rate of reproduction of the whole population. The rate for the whole population is calculated on the assumption that every female produces one offspring each year. Thus the fecundity of an individual female is 1. Since there are only 50% females, the fecundity of the population is .5.

```
CF_BORN = N_WB * WB_FEC
```

D: Net Wildebeest Reproduction The number of new offspring that survive is the product of the number born times the calf survival rate which is computed in equation B. This net figure represents the rate of increase of wildebeest numbers per year due to reproduction.

```
WB_REPRO = CF_BORN * SPR_CF_SURV
```

E, E': Wildebeest & Alternate Prey Population Density This is obtained by dividing the number of individuals in the population by the area. This is only recomputed at each time step for wildebeest because the sizes of the other populations are assumed constant.

```
WB_DENSITY = N_WB / AREA_SGTI
AP_DENSITY = N_APREY / AREA_SGTI
```

F: Specific Predation Rate The number of wildebeest that are killed and eaten per unit time *per predator* is computed using a standard mathematical model of a predator-prey relationship. The capture coefficient is determinable indirectly from field data, and differs for each prey population. The various populations of alternate prey that exist in the Serengeti ecosystem, are all lumped together as if they were a single species. The coefficients used are

for zebra, which are representative of these species.

```
SPR_PRED_WB =  (WB_CAP_CF * WB_DENSITY) /
               (1  + (WB_HTIME * WB_CAP_CF * WB_DENSITY)
                   + (AP_HTIME * AP_CAP_CF * AP_DENSITY) )
```

G: Wildebeest Eaten The total number of wildebeest killed due to predation at each time increment due to all predators is equal to the simple product of the number of wildebeest times the predation rate from the previous equation. This can be viewed as the rate of wildebeest mortality due to predation. It contributes to the net rate of change of wildebeest numbers.

```
WB_EATEN = N_PRED * SPR_PRED_WB
```

H: Wildebeest Mortality (ignoring predation): The number of wildebeest that die due to 'natural' causes[4] is equal to the number of wildebeest times the specific rate of wildebeest mortality (which is the difference between 1 and the specific rate of survival).

```
WB_DIE = N_WB * (1-SPR_WB_SURV)
```

I: Wildebeest Population This is the only variable which depends on its value in the last iteration. Thus, it corresponds to a single differential equation for representing the model.

```
N_WB = N_WB - WB_DIE - WB_EATEN + WB_REPRO
```

Note that the wildebeest population size is the only state variable whose value is computed at each time increment and whose values over time are of interest. The value of every other variable may be thrown away after each iteration, and recomputed the next time around. We use the term *state variable* is a specialised manner; we define it in § 2.4.1.2.

Comments

In the paper [Hilborn & Sinclair, 1984], there is a considerable amount of information about the ecological system being modelled; assumptions of the model couched in ecological terms, and the [conceptual] model itself. However, even for this very simple example, understanding the model was a considerable effort. We did not have the benefit of a Fortran program to start from, only a written

---

[4] Predation is of course perfectly natural. We mean to refer to things like old age, sickness, etc.

```
SUBROUTINE WBMODEL(NYEARS)
C Parameters and Constants
C
C AREA_SGTI       Total area of Serengeti being considered
C GRS_WT          Amount of dry season grass produced per year
C SPR_CF_SURV     Specific rate of calf survival per year
C WB_HTIME        Handling time of wildebeest
C AP_HTIME        Handling time of alternate prey
C WB_CAP_CF       Capture coefficient for wildebeest
C AP_CAP_CF       Capture coefficient for alternate prey
C AP_DENSITY      Population densities for alternate prey
C N_PRED          Number of predators
C N_APREY         Number of alternate prey
C DRY_SSN_RAIN    Amount of dry season rainfall per year
C WB_FEC          Wildebeest fecundity
C                    (annual specific rate of wildebeest reproduction)
C WB_INIT         Initial number of wildebeest

C Variables
C
C CF_BORN         Total number of wildebeest calves born in a year
C WB_REPRO        Total annual number of wildebeest calves survived

C SPR_PRED_WB     Annual Specific rate of predation of wildebeest by predators
C WB_DENSITY      Population density for wildebeest
C WB_EATEN        Total annual number of wildebeest eaten by predators

C SPR_WB_SURV     Annual Specific rate of wildebeest survival
C                    (ignoring predation)
C WB_DIE          Total annual number of wildebeest dying
C                    (ignoring predation)

C N_WB            Number of wildebeest

C T               Time in years

C Tabulation of Output Results
C Up to 10 output variables may be recorded for up to 50 time steps
C E.g for this example, we store values for the number of wildebeest
      and number of wildebeest eaten for each year.  This is done as follows:
C OUTPUT(1,T) Value of N_WB at time T
C OUTPUT(2,T) Value of WB\_EATEN at time T

DIMENSION OUTPUT(10,50)
INTEGER T
```

'C' is the comment character.

Figure 2–2: Simple Wildebeest Model: Documentation

```
C Initialisation of Parameters and State Variables

DATA DRY_SSN_RAIN    /250/      % mm
DATA SPR_WB_SURV /.95/
DATA WB_FEC      /.5/
DATA AREA_SGTI   /1000000/
DATA WB_HTIME, AP_HTIME       /.08, .05/   % Empirically determined
DATA WB_CAP_CF,AP_CAP_CF      /317, 100/   % Empirically determined
DATA WB_INIT, N_PRED,  N_APREY    /?,?,?/

N_WB = WB_INIT

C Equation E'
AP_DENSITY = N_APREY/AREA_SGTI

C Equation A
GRS_WT = 200 + DRY_SSN_RAIN*2

C Equation B
SPR_CF_SURV = (GRS_WT * .05) / (75 + GRS_WT)

DO 100 T=1,NYEARS

C Equation C
CF_BORN = N_WB * WB_FEC

C Equation D
WB_REPRO = CF_BORN * SPR_CF_SURV

C Equation E
WB_DENSITY = N_WB/AREA_SGTI

C Equation F
SPR_PRED_WB = (WB_CAP_CF * WB_DENSITY) /
     0            (1 + (WB_HTIME  * WB_CAP_CF * WB_DENSITY)
     0               + (AP_HTIME  * AP_CAP_CF * AP_DENSITY) )

C Equation G
WB_EATEN = N_PRED * SPR_PRED_WB

C Equation H
WB_DIE  = N_WB * (1-SPR_WB_SURV)

C Equation I
N_WB = N_WB - WB_DIE - WB_EATEN + WB_REPRO

C Tabulate raw results
OUTPUT(1,T) = N_WB
OUTPUT(2,T) = WB_EATEN

100 CONTINUE
RETURN
```

'C' is the comment character. We have inserted a few extra comments using % at the end of some lines of code.

Figure 2–3: Simple Wildebeest Model: Fortran Code

description. We were especially careful to document the model in as clear and informative a manner as possible (see figure 2–2). Even so, the Fortran program only contains the simulation model; none of the ecological information or modelling assumptions are represented in any useful sense. What useful information there is in this regard, exists only in the form of comments describing the intended meaning of the variables. It is neither available to nor understood in any sense by any program intended for processing it. This is a great obstacle for future use or modification (as described in § 2.2).

Focusing on this example for inspiration, in the remaining sections of this chapter we discuss the range of ecological and modelling concepts that our system should be able to represent. In chapter 4 we consider mechanisms for representing and reasoning about modelling assumptions. These form the bridge between ecological information on the one hand, and the model on the other hand. This constitutes a basis for facilitating model comprehension in a simulation environment.

First however, we represent the Fortran version of the model equations in a logical formalism similar to that used in [Bundy & Uschold, 1989] and based on the typed lambda calculus [Barendregt, 1985] (see figure 2–4). This version of the equations is an almost direct translation, and as such there are few advantages to be gained by doing only this. We do this here because the typed lambda calculus will serve as the basis for all the representations described in this thesis.

2.3.3 A Logic Representation

The details of our formalism are presented in chapter 5. Here, we outline the key features. Everything has a type. Three important types are *phys_obj*, *real*, and *time*. The former is the type of physical objects; *real* denotes the set of real numbers. We use the notation $X:T$ to denote that the type of X is T (or equivalently, X is an instance of T). Instances of *time* denote specific periods of time of some length; *e.g.* $yr(1):year$ denotes year one in the simulation. We also have types for different kinds[5] of physical objects; *e.g.* *animals* is the type

[5] To avoid mixing technical and non-technical usage of the same words we use 'kind' in a non-technical sense where 'sort' or 'type' would otherwise be appropriate; the latter are reserved for technical use.

of animals, *sheep* is the type of sheep. Types are partially ordered by a subtype relation denoted by $\sqsubset$; *e.g.* $year \sqsubset time$, $sheep \sqsubset animal$.

Functions are used to represent attributes like *sex*, *weight*, and model variables like *n_wb*. For example, if *shp*:*sheep*, then $weight(shp, yr(1))$ represents the weight of *shp* in the first year. *weight* is a function from physical objects and times to real numbers. Model variables only vary over time, not physical objects. Thus, *n_wb* is a function from *time* to *real*. We use the notation $\times$ for cross product, and $\mapsto$ for function mapping. The types of *weight* and *n_wb* are:

$$weight : phys\_obj \times time \mapsto real$$
$$n\_wb : time \mapsto real$$

The language also uses a variety of higher-order functions, λ–abstraction, quantification, and other things (details in chapter 5).

The types of the constants and functions used in the logic representation of the model in figure 2–4 are given in table 2–1. With the exception of *rate*, which is needed to represent differentiation, the primitives in the logic are quite straightforward. All the constants are of type *real*. For now it suffices to know that *rate* is a second order function from one unary function to another unary function of the same type (usually $time \mapsto real$).

Because they are virtually identical (differences noted below), we certainly do not mean to suggest that the logic representation is any better than the Fortran version. The model variables carry the same amount of explicit ecological information (*i.e.* none) in both cases. The purpose of this exercise is to help make the following discussion more precise, and to enable more direct comparison with the enhanced logic representation that we describe in chapter 5.

There are two notable differences between our two representations of the equations. One is that we explicitly distinguish between model variables and constants by making all the model variables functions from time to reals. With good reason, the time dependency is only implicit in the program. To have it otherwise would require saving every value of every variable at every time step. Most of this is unnecessary and thus would be very wasteful of space. The output array is used to store exactly what is needed. The other difference is that the program uses an assignment statement rather than a differential (or difference) equation.

This completes the discussion of the Serengeti example per se. Next we explore what is required to represent ecological simulation models.

$$\mathbf{E'}: \quad ap\_density(yr(I)) = n\_aprey/area\_sgti \tag{2.1}$$

$$\mathbf{A}: \quad grs\_wt(yr(I)) = 200 + dry\_ssn\_rain \cdot 2 \tag{2.2}$$

$$\mathbf{B}: \quad spr\_cf\_surv(yr(I)) = \frac{grs\_wt(yr(I)) + .05}{75 + grs\_wt(yr(I))} \tag{2.3}$$

$$\mathbf{C}: \quad cf\_born(yr(I)) = n\_wb(yr(I)) \cdot wb\_fec \tag{2.4}$$

$$\mathbf{D}: \quad wb\_repro(yr(I)) = cf\_born(yr(I)) \cdot spr\_cf\_surv(yr(I)) \tag{2.5}$$

$$\mathbf{E}: \quad wb\_density(yr(I)) = \frac{n\_wb(yr(I))}{area\_sgti} \tag{2.6}$$

$$\mathbf{F}: \quad spr\_pred\_wb(yr(I)) = \frac{wb\_cap\_cf \cdot wb\_density(yr(I))}{\begin{array}{rl} 1 & + \ (wb\_htime \cdot wb\_cap\_cf \cdot wb\_density(yr(I))) \\ & + \ (ap\_htime \cdot ap\_cap\_cf \cdot ap\_density(yr(I))) \end{array}} \tag{2.7}$$

$$\mathbf{G}: \quad wb\_eaten(yr(I)) = n\_pred \cdot spr\_pred\_wb(yr(I)) \tag{2.8}$$

$$\mathbf{H}: \quad wb\_die(yr(I)) = n\_wb(yr(I)) \cdot (1 - spr\_wb\_surv) \tag{2.9}$$

$$\mathbf{I}: \quad rate(n\_wb)(yr(I)) = -wb\_die(yr(I)) - wb\_eaten(yr(I)) + wb\_repro(yr(I)) \tag{2.10}$$

This set of equations is an alternate to the Fortran representation in figure 2–3. The two representations are nearly identical. One difference is that the proper model variables are time-dependent functions in the logic representation. Another difference is that the final equation is a differential equation rather than an assignment statement.

Figure 2–4: Serengeti in Logic

Category	Term	Type
Proper Model Variables:	n_wb :	$year \mapsto real$
	grs_wt :	$year \mapsto real$
	$ap\_density$:	$year \mapsto real$
	spr_pred_wb :	$year \mapsto real$
	spr_cf_surv :	$year \mapsto real$
	cf_born :	$year \mapsto real$
	wb_repro :	$year \mapsto real$
	wb_die :	$year \mapsto real$
	wb_eaten :	$year \mapsto real$
	$wb\_density$:	$year \mapsto real$
Parameters:	$area\_sgti$:	$real$
	dry_ssn_rain :	$real$
	spr_wb_surv :	$real$
	wb_cap_cf :	$real$
	ap_cap_cf :	$real$
	wb_htime :	$real$
	ap_htime :	$real$
	wb_fec :	$real$
	n_pred :	$real$
	n_aprey :	$real$
Time:	yr :	$naturals \mapsto year$

This table gives the formal types of the constants and functions used in figure 2–4. The details of the logic language are given in chapter 5.

Table 2–1: Types

2.4 Simulation Modelling Information

Here we give a synopsis of the nature and range of modelling information that we require for representing runnable models based on differential equations. Modelling information relevant to the process of constructing models is discussed in chapters 3 and 4.

As the above example shows, the representation of ecological simulation models need not contain any ecological information at all. One of the major claims in this thesis is that it is *absolutely essential* that ecological information related to the simulation model is represented if we are to facilitate model comprehension. Designing appropriate representations to support this has been a major part of this research project.

2.4.1 Model Variables and Parameters

In this section we define and classify model variables and parameters. Amongst the ecological modelling community, there is not a universally agreed set of terminological conventions for the different kinds of variables and the usage of the term 'parameter'. In § 2.4.1.1 we introduce the terms that we require and say exactly what we mean by them. In § 2.4.1.2, we use this to define various kinds of model variables precisely.

2.4.1.1 Terminology

We use the term *model variable* to refer to 'something' which has a value which may be used to compute another 'something', or which may be computed using another 'something'. This conforms to the common use of the word 'variable' in a computer program which is ambiguous on the issue of whether its value is constant or varying. So in our usage, a parameter is a kind of model variable.

We avoid the temptation to use either of the words 'quantity' or 'attribute' for this. 'quantity' implies numeric values. A variable representing the attribute 'colour' would have values like 'blue' and would not sensibly be referred to as a quantity. We do not use 'attribute' because of the need to distinguish the ecological and modelling levels of information. We reserve the term 'attribute' for the ecological level. Ecological variables are the idealised versions of [ecological]

attributes. Attributes are at the ecological level, variables are at the simulation modelling level. Analogously, the simulation model as a whole, is the idealised version of the ecological system.

The word 'variable' is agnostic about values and meaning in exactly the way we require. It can have anything as a value, and it can mean anything. We describe attributes in § 2.5.5.

We identify a notional continuum of variability on which there are four identifiable points. We start with the strongest sense of being constant: there is a kind of thing that is referred to as "a constant", (*e.g.* the gravitational constant). Its value is usually fixed by laws of nature. We shall refer to this as a *proper constant.* Its type would almost always be *real.* The only exception would be for non-numerical variables (*e.g.* one to represent the colour of something). For the remainder of § 2.4, we generally assume that everything is numeric. Thus when we say something is *always* of type *real*, or $time \mapsto real$ this is not strictly true. In § 2.5.5.3 we discuss both numeric and non-numeric values. In chapter 5 we give details of how these are represented.

Next, we have something which for different runs of a model may have a different value, but is initialised (not computed) and thus remains constant for a single run (*e.g. n_pred, wb_fec* in our example model). We adopt the conventional usage and refer to this as a *model parameter.* We do not distinguish between parameters and proper constants, although it could easily be done. Providing this would require a minor alteration and would provide a minor benefit. The type of these is also *real*, although in the ecological system, what the parameter represents could be represented as a function. For example, *n_pred* represents the attribute *number* which is a function from sets of physical objects and times to numbers.

The third case consists of something which is computed using some equation, but the inputs to the equations ultimately depend only on parameters and proper constants. This means that it is constant for the duration of the simulation (*e.g. grt_wt* in the example model). If *dry_ssn_rain* were not a parameter, but rather changed from year to year to more closely reflect reality, *grs_wt* would not be constant. Thus, the fact that it is constant is incidental. *grs_wt* is constant, it is certainly not 'a constant'; nor would it normally be viewed as a model parameter, because it is computed rather than initialised. We therefore do not refer to things

in this third situation as constants at all, but rather as *proper model variables* or *proper variables* for short. The types are of the form: $time \mapsto real$.

The final kind of thing genuinely varies over the course of a single simulation (*e.g. wb_density*). We also refer to these as proper model variables. Practically, it is useful to distinguish between these and the previous category because the former only need to be computed once. Our formalism supports this implicitly, and makes use of the distinction. Nevertheless, both are the same type (*i.e.* $time \mapsto real$) and we usually do not distinguish them.

2.4.1.2 Kinds of Model Variables

Modellers (ecological and otherwise) have identified a number of different kinds of variables. We list and define these according to our understanding of conventional usage. We make certain additional distinctions. Except for parameters which are of type *real*, all model variables are of type: $time \mapsto real$.

state variables: model variables whose values are computed over each time step in the simulation by incrementing and/or decrementing the value from the previous iteration. The inc/decrements are determined by the partial rate variables.

e.g. n_wb

partial rate variables: model variables which are equal to the partial rate of change of the value of some state variable. The rate may be positive or negative. These usually represent a specific effect of some process.[6]

e.g. wb_eaten represents the rate of decrease in numbers of the wildebeest population due to predation.

net rate variables: model variables which are equal to the net rate of change of some state variable. The net rate is equal to the summation of all the partial rate variables for a particular state variable.

e.g. rate(n_wb) is the net rate of change of the number of wildebeest which increases due to reproduction and decreases due to mortality and predation.

exogenous variables: model variables that change over time, but do not depend on any model variable.

---

[6] The use of the word 'partial' here is to contrast with 'net'. It is not to be confused with partial differential equations, a quite different concept.

e.g. the example model has none, but *dry_ssn_rain* might be if a different value was read in every time step.

intermediate variables: model variables which depend on (*i.e.* are computed from) any other model variable but are not state or rate variables (net or partial). They will 'happen to' be constant if they neither depend on state nor exogenous variables.

e.g. grs_wt

parameters: variables that are constant for a run of the simulation.

We distinguish three kinds of variables which are not distinct from the above ones, but are distinguished by different criteria. The first are *attribute variables.* These represent attributes of ecological entities. For example, *n_wb* represents the attribute *number* of the wildebeest population; *grs_wt* represents the weight of the grass. Attribute variables may be any of the above kinds except for partial rate variables. For example, *n_wb* is a state variable; *grs_wt* is an intermediate variable.

Next, we have *effect variables.* They are defined to be variables that represent effects of processes. For example, one effect of the process of predation is to reduce the number of wildebeest. The effect variable that this gives rise to is *wb_eaten.* Effect variables will usually be partial rate variables. However, when the process is modelled in a way other than to increment or decrement some state variable, they are more likely to be intermediate variables but could also be exogenous variables or parameters. For example, the variable *grs_wt* can be viewed as an effect variable representing the effect of the growth process which is to increase the weight of the grass. The fact that it is not a partial rate variable does not change the fact that this variable represents an effect of the growth process. In this case, the effect variable and the attribute variable *grs_wt* are identical. Similarly, the variable *dry_ssn_rain* is the model representation of the effect of the precipitation process which is to increase the amount of water in the Serengeti. It is a parameter in the example model, but could also be an exogenous variable.

The third additional kind of variable is *output variable.* These are distinguished by being printed out as part of the model output. These will normally be state variables, but might also be intermediate, or rate variables (net or partial). They cannot sensibly be exogenous variables or parameters because the former are model inputs and the latter are constant. Output variables are of particular importance

because they can drive the entire model elicitation process by a process akin to backward chaining, as in [Robertson et al, 1987; Haggith, 1990].

In chapters 5, 6 and 7 the role of these different kinds of variables will become clear. As we shall later see, in spite of all these distinctions, in the object-level logic, there are only two basic types of model variable (see figure 2–1). Parameters are logical constants (usually of type *real*). Proper model variables are unary functions from time to some value space (*e.g.* $year \mapsto real$).

2.4.2 Equations

We distinguish two kinds of equations: differential, and all the rest (called non-differential). In figure 2–4, only equation 2.10 is a differential equation. There is by definition exactly one differential equation for each state variable in the model.

2.5 Ecological Information

In this section we explore the nature and variety of ecological concepts that must be represented. We begin by analysing the example model and note important missing ecological information. We outline our conceptual modelling framework for describing ecological systems and identify the requirements for our formalism. In keeping with the goal of minimising conceptual distance without an undue increase in the processing required to interpret the high level formalism, this conceptual model will map onto our formalism in a fairly direct way.

2.5.1 Missing Ecological Information

There is a great deal of ecological information implicit in the model representation, most of which is in the names of the variables. They imply the existence of various entities, attributes, processes, etc. However, this information resides only in the minds of the creator of the names; others can only guess their meaning. A computer system equipped only with this representation does not have access to it and thus can not make use of it.

For example, consider the two variables *n_pred,* and *n_wb.* We know that these represent the same attribute *number* [of members] of two populations. Fur-

thermore, populations are special kinds of entities that behave differently from individuals.

There are many simplifications and assumptions that are evident from the discussion, but are nowhere represented. These are crucial to model comprehension which enables the model to be safely and/or conveniently used and/or extended. Yet the representation does not support any way to record this information. Below we discuss some of the important ecological information that is not represented.

Entities and their Attributes

The representation only caters for variables and values; it does not capture the fact that the two variables *n_pred,* and *n_wb* represent the same attribute (*i.e. number*). Entities and their attributes are not distinguishable in this representation.

Sets and Substructure

In ecological systems, individual entities are frequently grouped together to form collective entities, or conversely, they are subdivided into components in some way. We use the term *substructure* to refer to any such structure defined among entities.

There is much potentially relevant substructure in the example ecological system. There are two important reasons for representing this even though it is not used in the simulation model. First, to facilitate model comprehension, we can explicitly document how and whether the substructure in the system is manifest in the model. Second, an extended version of the model might use the substructure. We note three examples of substructure which are *not* discussed explicitly in the paper [Hilborn & Sinclair, 1984].

First, there is no explicit notion of a population, or set of entities distinct from individuals. There are many advantages that derive from making this distinction; these are discussed in § 2.5.4.

Next, the wildebeest population has no age classes, despite the apparent existence of calves. This is glossed over due to the fact that the time scale of the model is years. There is no model variable representing the number of calves analogous to the variable for the number of the whole wildebeest population. Rather, a computation is done which concludes how many calves survived to the end of the year. At the end of the year, there are no longer calves, only yearlings and adult wildebeest. There is also no distinction (in the simulation model) between

yearlings and adults, even though the fecundity is notably different for yearlings and adult females. This simplification is a good first approximation because there are relatively few yearlings.

Thirdly, the wildebeest population is not subdivided into male and female. For this modelling exercise, the only important difference between the male and female wildebeest is that only females reproduce. As a simplification, this was ignored in the model.

We note four examples of substructure which *are* discussed in [Hilborn & Sinclair, 1984], but which are still ignored in the simple model.

First, the alternate prey population is a composite population consisting of several species including zebra, gazelle, and others. None of these are separately specified. They are modelled as a single population; the parameters used are those for zebra. So, the model is as if there was only one species of alternate prey, but the number of zebra in the model was artificially increased to represent the other species.

Next, the predator population is also a composite population; it consists of both lions and hyena. For the purpose of the model, a hyena is considered as 2/5 of a lion.

Thirdly, the time structure is simplified. The discussion in [Hilborn & Sinclair, 1984] specifically discusses events on a monthly scale. For example, wildebeest give birth in February. The specific months of the dry and wet seasons are given. All this is ignored in this model, although it is true of the ecological system being modelled.

Finally, the Serengeti itself is divided into 3 spatial zones which is relevant for migration.

Processes

The representation of processes is bound up in the equations. Nowhere does it explicitly say that wildebeest participate in the processes of reproduction, mortality, predation, etc.

There was an extended discussion of migratory behaviour as well. This process is ignored in the model. This constitutes a relevant modelling decision which should be recorded.

Summary: Missing Ecological Information

We have identified two major shortcomings with the current representation. First, the similarity between similar concepts is not captured. Second, no modelling decisions are represented. The advantages that derive from representing this missing information explicitly are many and varied. In chapter 4 we cover these in detail; here we list just two:

- Model Comprehension: in conjunction with general purpose machinery, documentation of the meaning of model variables in ecological terms is automatically generated.
  e.g. in ELK, the variable *n_wb* in the example model translates to a slightly terser version of:
  "The attribute 'number' of the entity 'wb_pop', a collection of wildebeest"
- Consistency Checking: ecological consistency can be ensured by the system because it 'knows' that the attribute *number* only applies to sets of entities not to individual entities. So while an individual animal, for example may have a colour and a size, it does not by itself have a 'number' attribute.

2.5.2 Fundamental Ecological Concepts

Ecological modelling is concerned with representing the state of some ecological system and how it changes over time. There are many possible metrics for the state; in most cases, such a measure can readily be viewed as a value of an attribute of some type of entity. So, insofar as describing ecological systems is concerned, we shall adopt an entity-attribute-value view of the world. This view serves as a suitable starting point for our analysis, but it only deals with describing the static state of an ecological system. We also need to incorporate the notion of process. For our purposes, we shall assume that an *ecological process* is something which causes the values of attributes of entities to change. Such attributes are often, but not always modelled as state variables. In § 2.5.6 we consider this and other details about processes.

Besides entities, values also have attributes which characterise them. For example, numbers may be even, odd, countable, etc. Attributes also have attributes such as whether or not the value space over which they range is ordered. However these secondary attributes would never be used to describe an ecological system

directly. They are concerned with information at a different level. To make the required distinction, we introduce the term *ecological attribute*; it refers to attributes that could possibly serve to characterise an ecological system. All other attributes are called *non-ecological attributes.* From a practical viewpoint, they are are distinguished from other attributes in that only ecological attributes give rise to model variables [including parameters, of course], *or* may be used to define substructure on other entities. Sex is an attribute which would rarely be represented as a model variable, however it may often be used to subdivide a population. We refer to entities that have ecological attributes as *ecological entities.* We also draw a logical distinction between values of ecological attributes (*e.g.* male), and values of non-ecological attributes (*e.g.* ordered). The former are called ecological values. We refer to non-ecological attributes (*e.g.* even, orderedness) as *properties.* Their [non-ecological] values are called *property values.*

As noted in § 2.1, our conceptual modelling framework is based on processes, entities, substructure, attributes, and values, (PESAV). Virtually everything that we shall be concerned with regarding the representation of ecological information is related directly or indirectly to one or more aspects of these five concepts. The terminological convention that 'X' is short for 'ecological X' holds for 'process', 'entity', 'attribute', and 'value'.

2.5.3 Ecological Entities

Traditionally, ecological entities in ecological systems include animals and plants which are naturally represented as taxonomies. There are many other types of entities such as bodies of water, mountains, substances (*e.g.* soil, nutrients). Mostly we are concerned with physical objects; but there are other entities as well such as places; *e.g.* it is reasonable to view an ecological system itself as an entity. Substances are discussed in § 2.5.5.

Usually, entities do not find explicit representation in ecological models; only their attributes do. Three of the more important and direct benefits that result from representing entities explicitly are:

- to reduce conceptual distance. By 'talking' about ecological entities, users are better able to interact with the system in a manner close to how they are thinking about their problem.

- to facilitate model comprehension through automatic documentation facilities as noted in § 2.5.1.
- used in inheritance machinery to help identify and constrain the possible choices of attributes for entities which ultimately leads to model variables and parameters

Also, some models include the creation and destruction of entities. The current version of ELK allows users to create and destroy entities, as part of describing the ecological system to be modelled. However, there is no way to do this in the simulation. This is a useful feature which is relegated to future work.

2.5.3.1 Average Members

It is common in ecological modelling to invent and reason about the average member of a set of individuals as if it existed. For instance, an ecologist might refer to "the average sheep in some particular flock". It would be endowed with the average properties of each individual in the flock (*e.g.* weight, height). In the example model the capture coefficient is not really an attribute of any particular animal. It may be viewed as an attribute of a fictitious 'average member' of a set of animals. An equally valid view is to take capture coefficient to be an attribute of the population. This has the advantage of not requiring explicit representation of fictitious entities. But it does require a mechanism for representing populations and more generally, sets. The requirements relating to sets and substructure are discussed in the next section. In § 6.7.1 we describe a facility for reasoning about average members.

2.5.4 Sets and Substructure

In ecological systems, individual entities are frequently grouped together to form collective entities, or conversely, they are subdivided into components in some way. We use the term *substructure* to refer to any such structure defined among entities. Note that grouping and subdividing are really alternate views of the same substructure. Consider a flock of sheep. This might be viewed as grouping if there were a number of explicit individual sheep being put into a single flock. Alternatively, it could be viewed as subdividing a flock into individuals. For this reason, we will usually simply talk about substructure rather than grouping or

subdividing. We identify three related but logically distinct ways that substructure can arise.

1. An individual may be a *member* of a set of similar individuals
   e.g. a flock of sheep
2. A set of individuals may be a *subdivision* of a larger set of similar individuals.
   e.g. the old male sheep in a flock constitutes a subflock of the whole flock.
3. An entity can be a *part* of a *composite*
   e.g. a branch of a tree

2.5.4.1 Terminology

The common English expressions that are used to denote these three situations reflect both the similarities and the differences between them. For instance, we would normally say that an individual sheep is a member (not a subdivision) of a flock[7]. Also, it is easy to view the sheep as a component or part of the flock, although we would not usually express it that way. We would speak of subdividing a flock of sheep into a male flock and a female flock. We would not say that the male sub-flock is a member of the larger flock, but we might refer to the sub-flocks as subdivisions of the whole flock. We might also say they are parts, or components of it. We would certainly not refer to a branch as a member or a subdivision of a tree, but rather as a part, or component of it.

Thus, in English, we have 'part', 'component', 'member', and 'subdivision' as common referents for the things that entities are composed of. 'Part' and 'component' are fairly general, and largely interchangeable, whereas 'member' and 'subdivision' have much more specific and indeed different connotations. We have chosen to use the words 'component' and 'whole', as the generic referents to describe all of the above examples of substructure relationships.

The reason for choosing 'component' instead of 'part' is that the 'part of' relation is commonly found in frame/object systems, but it does not usually refer to things like member and subdivision. For example, a section heading in [Blake & Cook, 1987] reads: "Elements of a Collection are not Parts". This is the

---

[7] N.B. member and subdivision are really just the common English words used to denote the set-theoretic concepts of element and subset.

point. But they *are* components, in my usage. We use 'part' when we wish to imply that the type of the part is fundamentally different from the type of the whole.

The reasons for choosing 'whole' instead of 'composite' are similar. The word 'composite' in English usually implies that the parts are different. A set of like entities would rarely if ever be referred to as a composite. Not surprisingly, the literature is not consistent in the use of 'composite' versus 'whole'. The term 'structured whole' was used in [Blake & Cook, 1987], though the authors possibly should have used 'composite'. In [Bunt, 1986] and [Leanard & Goodman, 1940], the term 'whole' is used as we use it here; but the authors use 'part', not 'component'. We use the term 'composite' when referring to a whole all of whose components are neither members nor subsets of it.

There are also many words commonly used to refer to the *process* of building up wholes from their component parts, or vice versa. We normally use 'group' or 'compose' for the former and 'disaggregate' or 'subdivide' for the latter. '(Dis)aggregation' is the term most frequently used in ecological modelling.

2.5.4.2 Uses

At this point, we are limiting the discussion to defining substructure as part of the ecological system description [conceptual model]. Deciding how and whether to incorporate the whole entities and their components in the simulation model is a separate issue. There are two fundamental reasons for defining any substructure corresponding to the two alternate points of view for substructure.

1. Subdivide an existing whole when there are differences between the components that are relevant to ones modelling objectives.
2. Compose a new whole when there are similarities among all the components or if differences are not important.

The decision to define substructure in the conceptual model is based on the same principle of whether anything goes in the conceptual model: its potential relevance to the modelling goals. The usefulness of including it is primarily to ensure model comprehension whether or not it is ultimately included in the simulation model.

The decision to subdivide may be taken if the components are subjected to different influences or processes in the ecological system. These differences can

be made explicit in the conceptual model. The decision to compose a new whole applies primarily to the member and subdivision cases. Several individuals might be grouped together into a set for the purpose of making some specification to each member of the set simultaneously. For instance, each may have a feeding rate which is the same within the group, but differs from other groups and/or individuals.

2.5.4.3 Sets

Sets of entities are extremely prevalent in ecological systems, (*e.g.* animal populations, forest stands). Although composed of entities, sets are also entities in their own right. An important requirement therefore is to support a variety of useful inferences related to sets. In chapter 5, we shall see that sets play a prominent role in our formalism, binding together the representation for substructure among other things.

Our formalism supports the ability to represent sets of entities without having to represent the individual members. This is necessary because in many population and forestry models, the individuals are not of interest and thus not directly represented. Although ignoring the individuals, we retain an explicit link with the type of entity that the set consists of. This facilitates being able to infer information about the attributes of the sets from information about the attributes of the member or subdivision types. This is described in § 2.5.5.

A somewhat different sense of not representing the individuals in a set arose in the example model discussed in § 2.5.1. The alternate prey population is a composite population consisting of several populations of different species. In the simulation model, the constituent populations are not represented. In this case the member populations are themselves represented as sets. This is usually referred to as aggregation by ecologists and is an important kind of modelling decision. While the *model* requires that no distinction be made between the different subpopulations, representing the distinction enables the recording of this modelling decision. To facilitate model comprehension, ELK caters for the recording of various kinds of modelling decisions including this one.

2.5.4.4 Composites

Composite entities are also of considerable importance in ecological modelling. Many forestry models are concerned with processes that affect the trunk, branches, and sometimes even needles [Mitchell, 1975]. A sheep model might be concerned with wool. A composite need not be an individual, but a more complex entity like say, an ecosystem, or subsystem.

Note that in a forestry model, while needles and branches may need to be represented as ecological entities which have their own ecological attributes that describe the model, there may be no specific need to model the trunk this way. The only interest in the trunk may be the amount of wood it contains. This might more naturally be represented as an attribute of the tree. In this case, having the trunk would be superfluous, and the formalism would not necessarily 'know' about wood. Similar comments apply to the wool of a sheep.

2.5.4.5 Uniformity

A final point is that the same model may contain all three kinds of substructure. Because, at one level of abstraction each of the three kinds of substructure are the same, there are advantages of having a single a general mechanism which captures this. One is avoiding redundancy; for example in [Bunt, 1986], Bunt notes that many of the logical properties of the part-whole relation are the same as those of member-set and subset-set. There is no point in specifying the same axioms twice, once for set theory, and once for the part-whole relation.

A single component–whole relation encapsulating three kinds of substructure contributes to uniformity in the representation which in turn gives rise to uniformity in the user interface. This uniformity contributes to an overall simplicity and elegance rendering both the representation and the interface easier to understand. Of course, the general mechanism also needs to be able to make the distinctions when necessary. Such a mechanism is described in chapter 5.

2.5.4.6 Homogeneity

We can distinguish entities on the basis of whether they can have components *of the same kind.* We refer to such entities (or their types) as *homogeneous.* Sets (as opposed to individuals) are one example. A set of trees can have component subsets which are also sets of trees. Some individuals are homogeneous. For

example, a region can be subdivided into sub-regions; a quantity of water may be subdivided into other quantities of water. However, many kinds of individuals are non-homogeneous. For example, a tree can be subdivided up into roots, branches, etc. but not into other trees. Also, a lake may be subdivided up into quantities of water, each of which are homogeneous, but a lake cannot be subdivided into other lakes.

This distinction is important if we are to prevent users from specifying sub-structure that is inconsistent. For example, we can use this concept to determine whether overlapping should be allowed. Two sets may overlap, but two trees may not share a single branch.

The concept of homogeneity is related to but not the same as the distinction between continuous and discrete things. Most continuous things are homogeneous, but the reverse is not the case. For example, a region, or a quantity of water is continuous, but a set is not. Although continuous concepts arise frequently in the domain of ecology, we have found no need to go beyond representing the concept of homogeneity.

2.5.5 Attributes and Values

Because attributes are fundamental to ecological modelling, it is important to represent them properly as well. Our usage of the term 'attribute' accords very much with the dictionary definition:

> "a property, quality, or feature belonging to or representative of a person or thing [Hanks, 1980]"

We do however make some important distinctions that this definition fails to. One is between attributes, values, and predications. For example, a property, quality, or feature of a particular sheep might be that it is male, or that it weighs 80 kg. Formally, these might be represented as $sex(shp) = male$ and $weight(shp) = 80$ respectively (for the time being, we ignore the units). While the dictionary suggests that these expressions are themselves attributes, we refer to them as predications because they can be true or false. What we call the attribute is the thing that can have a value (*e.g.* sex, $weight$). We use the phrase 'E has A' as an abbreviated form of 'the attribute A applies to the entity E'.

Aside: Units are of major importance in the ecological domain, and proper support should be provided. However, because other things were of higher prior-

ity, we have not yet included this in our implementation. Although mechanisms for handling units and measures are widely available (*e.g.* [Bundy et al, 1979; O'Keefe, 1985]), it would constitute a significant effort to properly integrate these into our formalism and implementation.

2.5.5.1 Inheritance, Induced Attributes

The most basic inference facility we require is a standard mechanism for inheriting attributes from more general types of entities. For example, from the fact that physical objects have weight we can infer that sheep do too. There are other inferences that we have a need to support. For example, there are a range of commonly arising operations such as totalling, taking an average, or finding maxima and minima. We might wish to talk about the average weight of a set of sheep at some fixed point in time, or about the maximum weight for a single sheep over a period of time. We shall describe a general mechanism whereby the system can infer that 'average weight over a period of time' is an attribute that applies to sheep, and that 'maximum weight' is an attribute that applies to a set of sheep from the fact that the attribute weight applies to sheep. Furthermore the mechanism 'knows' that even though sex is also an attribute of sheep, 'maximum sex' is not an attribute because the possible values of the sex attribute are not ordered.[8] We call these *induced attributes*; they are implicitly defined. Note the intimate relationship between these attributes and the concept of sets.

This inducing of attributes is distinguished from standard inheritance in that rather than existing attributes being inherited by different entities, these are new attributes which are derived from existing attributes and they are 'inherited' by either the same or different entities. There are more complex concepts which can be viewed as induced attributes. These include equilibrium and threshold. In these cases, the induced attributes most sensibly apply to the ecological system, rather than to a more specific entity. Additional discussion of equilibrium and threshold is found in § 3.4 and § 6.5.1.1.

---

[8] Our patriarchal society notwithstanding.

2.5.5.2 Using Attributes

Another important distinction that we make is with respect to how attributes are used. We distinguish two roles that attributes play: *defining* and *describing*. Consider the attributes 'number of legs' and 'weight' that apply to sheep. The former is used in a defining capacity, has the value 4, and in principle, does not vary. The latter is used in a describing capacity and it may vary. We are primarily interested in the describing role for attributes. It is these that give rise to model variables which characterise the state of the ecological system. Our distinction between the defining and describing role of attributes is analogous to the *terminological* versus *assertional* distinction (T-Box and A-Box) in term subsumption languages [Patel-Schneider et al, 1990].

In their defining role, attributes are dimensions for classification. For example, a certain combination of values of animal attributes gives rise to various subclasses such as mammals and reptiles. These were identified and used by biologists in building 'the' animal taxonomy. This level of detail is not required for our system. In our formalism, the attributes are merely associated with the types of ecological entities. For example, the attribute *weight* applies to the type *phys_obj*. This helps characterise physical objects, but is not part of the definition of that type in the same way that a set of attribute value pairs may be. For example, we could define 'male lion' to be a special kind (*i.e.* subtype) of lion characterised by the value of the attribute sex being male. This is common in term subsumption languages [Patel-Schneider et al, 1990; Patel-Schneider, 1984] for which classification is a major feature. We assume that ecologists know what these types are and can reason about them. We view animals, mammals, sheep etc as primitive concepts. Thus, we are mostly not concerned with the definitional role of attributes.

There is one important exception which we cater for which arises in the context of disaggregation. For example, a population of sheep may be subdivided into subgroups according to sex, age, location, etc. An old male sheep can be viewed as a kind of sheep, but in an intuitively different sense than a sheep being a kind of mammal. Unfortunately, formalising this intuition is exceptionally difficult, if not impossible. It is to do with the issue of natural kinds, philosophical distinctions, etc. Fortunately, we do not have to worry about this; it is up to the ecologists to decide what the 'natural kinds' are that they are interested in. We describe a technique for representing substructure of this nature in § 5.6.6.3.

Note that although these two roles are quite distinct, one attribute may be used both to define and describe. The attribute weight would commonly be used to *describe* individual sheep; each sheep would have a weight which could vary over time. Weight could also be used to *define* size categories for sheep, (*e.g.* large, medium, small). A sheep population could be subdivided according to these categories in exactly the same way that it may be subdivided according to the categories (male, female) for the sex attribute. Furthermore, the changing values of weight of the individuals might dynamically alter the members of the subdivisions. Although we do not cater for this latter form of dynamism, our basic mechanism supports this dual role of attributes. In § 2.5.6 we discuss other dynamic behaviour that we do not support at this time.

2.5.5.3 Values

Values that attributes can have are many and varied. In ecological modelling, values for attributes used in the descriptive role are almost always numbers. For example, weight, height, biomass are all measured in numbers, although the units are different.

There are other kinds of values however, such as for the attribute 'colour' (*e.g.* red, green). These might be used in either a defining or describing role. Used in the defining role we might define notional kinds of sheep on the basis of their colour which is presumed not to change (black sheep, white sheep). The descriptive role would be appropriate where the colour of an entity changed, say a chameleon, or grass seasonally going brown and green. These could well give rise to model variables.

Every ecological attribute and proper model variable can be thought of as a function. Its range is called a *value space.* There are a number of important issues which can be used to classify value spaces. These are covered in great detail in [O'Keefe, 1985] which describes a powerful and general mechanism for handling entities, attributes, and value spaces which incorporates the notion of units and measures. We do not require all that power because the issues which concern us differ. Our conceptual model for value spaces is largely derived from that work, although it is greatly simplified. However, the functionality we provide is rather in excess of that provided by any other systems related to ecological modelling (see chapter 8).

The distinctions that we have found useful regarding value spaces are as follows:

- ordered versus unordered
  e.g. {small, medium, large}, versus {red, blue, green}
- is addition defined
  e.g. yes for real numbers, no for {small, medium, large}
- finite versus infinite *e.g.* yes for natural numbers, no for $\{male, female\}$

We note some examples of how these properties are used. It does not make sense to apply maximum to an unordered value space. Nor is it possible to compute an average of an attribute unless addition is defined on its value space. Logically, it is possible to define substructure which gives rise to an infinite number of subdivisions. For example, we could use the real numbers to define infinitely many kinds of sheep on the basis of the value of the weight attribute. However, we disallow this because it is unlikely to be useful.

2.5.5.4 Values and Entities

Let us summarise what we have so far. Only ecological attributes are used to describe the state of an ecological system and thus can give rise to model variables (we do not characterise a sheep as being ordered or not). Only ecological attributes may be used to define substructure on ecological entities. Only ecological attributes have ecological values. Only ecological entities have ecological attributes. Values and value spaces have non-ecological attributes which we call properties. Values themselves are entities, as are value spaces.

In our formalism, ecological attributes and their values are all handled using a single uniform mechanism. However, properties and their [non-ecological] values are not represented explicitly as attributes and values. Rather they are handled in a somewhat ad-hoc manner. Figure 5–2 shows a hierarchy of the fundamental types of entities.

One final distinction that is significant is that ecological values are never ecological entities (although they are entities). This is a departure from standard representations where for example 'part of' is represented as an attribute. For a tree, it might have a value which was a branch, another ecological entity. In [O'Keefe, 1985], this distinction is also made, but described using different terminology. Two kinds of attributes are distinguished: those whose values are other entities, and those whose values are not. The former are called components, the

latter properties. We use a specialised representation for components and thus have only one kind of [ecological] attribute.

2.5.5.5 Multiple Value Spaces for the Same Attribute, Equivalence Classes

The main argument for distinguishing between the ecological and modelling levels is to enable idealisation decisions to be recorded thus facilitating model comprehension. We illustrate this with the important case of attributes and model variables. Consider the attribute *weight*. Its values are always positive reals. Often, model variables representing the weight of some entity would use this value space, as a default. However, for some entities it may desirable to idealise weight in three categories, say small, medium, and large. For yet other entities, it may be desirable to have finer resolution for representing weight values (eg adding 'very small', and/or 'very large'). Thus, the same attribute may be idealised differently for the different entities to which it applies, *possibly in the same model.*

To record these idealisation decisions, it is necessary to have *a separate representation for the real attribute and for each possible use [of the attribute] which gives rise to a model variable.* We have the true value space, and any number of idealised ones. The total number of idealised value spaces is equal to the number of model variables and parameters that the attribute gives rise to in the model.

Note that the value spaces used by the model variable must be logically related to the original value space. So, in the above example, small, medium, and large might constitute a partition of the set of positive reals. In general, each value in the derived space must constitute an equivalence class in terms of the original one. For example, it might be that $large = [100, 300]$. If *large* was used as a dimension for defining substructure on a flock of sheep, then the large subflock is characterised by each member (whether explicit or not) having the value of the weight attribute be in the interval $[100, 300]$.

2.5.5.6 Substances

In ecological modelling, substances are very important. However, they are not usually treated as entities whose attributes are used to describe the ecological system. They are [conceptually] modelled as attributes themselves, as in the wood example (§ 2.5.4.4). These attributes denote the amount of some substance

present in some other entity. For example if we were interested in DDT, it would almost always be modelled by some variable which denoted the amount of DDT in some entity, say a population, or body of water. The same is true for things like biomass, nutrients, minerals etc. Thus, in our conceptual model, we represent these as ecological attributes. An alternate more general way to represent these is discussed briefly at the end of § 7.3.2.

2.5.5.7 Summary: Attributes and Values

Attributes and values are central to our concerns and we have taken great pains to represent them in as rich and general way as possible. We note three major requirements:

- to support inheritance and induced attributes
  e.g. 'average weight' is induced from 'weight'
- to support the dual role of attributes
  i.e. defining and describing
- to support multiple value spaces for the same attribute
  e.g. {small,medium,large} and positive reals.

In designing techniques for supporting these requirements, our guiding philosophy was to minimise redundancy in the representation by making explicit connections between similar concepts and keeping the number of primitives relatively small . For example consider *number* as discussed in § 2.5.1. We shall not have to separately represent the value space for 'number of wildebeest', 'number of alternate prey'. This is done once for the attribute *number* which applies to all sets of objects. This gives rise to a variety of advantages which are described in chapter 4. One of these is the ability to automatically document the model.

2.5.6 Processes

Ecologically, a process is some kind of activity in a system. We shall limit this discussion to processes which explicitly effect some measurable change in the ecological system. For example, grazing causes a transfer in biomass from the grazed to the grazers. Predation causes the number of individuals in the prey population to decrease, and the biomass of the predator population to increase. It may also cause the amount of DDT in the predator population to increase. These are all

ecological truths. If these processes are present in a system, they may or may not be modelled explicitly in these terms, if at all.

Processes are very general, and we wish to capture as much of this as possible. However, they are almost infinitely varied and we must make simplifications. For our conceptual modelling framework, we adopt an attribute-based view of processes. That is, the measurable changes caused by a single process are represented by changing values of one or more attributes of one or more ecological entities. The nature of the change is presumed to be in accordance with some implicit laws of nature as opposed to haphazardly. Not all processes are easily or adequately characterisable using this view. Competition is one example. It is not clear what (if any) attributes of two predators changes as a result of their competing. This process indirectly affects the process of predation which directly affects the attribute biomass among others. Other processes that we cannot adequately represent are those which result in structural changes to the participating entities (*e.g.* melting ice, burning) or their creation and destruction.

Creation and destruction arises in the context of population dynamics systems, where we are primarily interested in population sizes in numbers. The usual case is when individuals are not modelled; then, the attribute *number* is all that is required to represent the process. However, if individuals *are* being modelled, then this may result in the explicit creation or destruction of entities which is not directly representable as changing values of attributes of entities. The same attribute 'number' is still the one of primary interest, but in this case, the process is not wholly characterised by this attribute. The attribute is an indirect measure of the effect of the process which directly creates and destroys entities.

2.5.6.1 Comparisons with System Dynamics

We incorporate and extend the view of processes embodied in system dynamics methodology (see appendix B). See [Wolfe et al, 1986] for a good discussion of the use of this methodology in the domain of ecological modelling. In that view, all processes model the changing amount of some stuff represented by a state variable that is transferred between ecological entities at each time increment. The stuff is often physical, but may be virtually anything that can be conceived with sufficient imagination to flow between entities (*e.g.* energy, money). Also, in a two entity process, it is assumed that the same stuff flows from one entity into

the other. However, these views are not always natural. Consider predation. It is perfectly natural to represent this as a flow of biomass from prey to predator; this is fine. However, if numbers are of interest, it is not possible to similarly view this process as a flow of numbers. 'Numbers of prey' is different 'stuff' than 'numbers of predators'. If modelled in the system dynamics formalism, this would require two separate flows involving the source and sink. Thus we would have to conceive of numbers of prey flowing into a sink, and numbers of predators flowing from a source. There are other possible ways that predation might be modelled. Some of these are:

- change in prey numbers, but predators numbers remain unchanged (or vice versa). This is exactly what is done in the Serengeti model.
- change in prey numbers, but predator numbers are not modelled, rather only the biomass of predators is of interest. (or vice versa).
- change in both biomass *and* numbers of both the predator and prey populations.
- any of the above additionally modelling the amount of DDT in either of the prey or predator populations or both.

In system dynamics terms, for a given process, separate flows are needed for each distinct 'stuff' that 'flows'. Thus, the simple analogy of a single unique flow for each process breaks down very quickly. A single process may involve many distinct flows; the fact that they correspond to the same process is not representable in the language of system dynamics. It would be preferable to be able to think of the process of predation between these two populations as a single notion which had possibly many effects. We shall define a more general representation which retains the explicit connection between the various effects of a single process. In doing so, we do not lose the expressive capability of system dynamics models. Instead, we retain it as a convenient shorthand for our representation when the flow analogy does hold.

Another limitation of system dynamics models is that a process necessarily has two participants, one which the stuff flows into and one which it flows out of. The fictitious entities *source* and *sink* are created when a process in fact only involves a single agent, as for mortality. While it is true that the vast majority of processes involve only one or two agents, there is no reason for this to always be the case. In our representation, the number of agents is open ended.

We now summarise the main extensions of the system dynamics view of processes that we have made:

- we give an explicit account of how the process is being modelled in ecological terms.
- a single process may have more than one effect and the connection with this process is explicitly recorded.
- for two agent processes, it is not *a priori* assumed that there will be a single conceptual substance which is transferred from one agent to the other
- the model variable that changes as a result of the process need not be incremented or decremented; any specification of a change is possible.
- following from the previous point, we enable processes to affect attributes with value spaces for which addition need not apply (ordered and unordered).
- the number of agents in a process is not fixed at two.
  - there may be arbitrarily many
  - when only one, no fictitious source/sink is required

2.5.6.2 Influences

One very important idea in ecological systems is that of *influences*. For instance we might say rainfall influences grass growth, or that rate of predation influences population size. Influence diagrams are a useful way for ecologists to organise their thoughts about the processes in an ecological system. Providing a facility for expressing this information can play an important role in reducing conceptual distance. In simulation modelling terms, an influence is idealised as a computational dependency between two model variables.[9]

2.5.7 Time

Finally, we require at least a rudimentary representation for time. This derives directly from the need to model processes. There are two cases where reasoning about time is required. The simpler is for computing averages, maxima etc of some attribute of some ecological entity over some time period. Based on some

[9] Influences are not included in the current implementation, but they fit in a natural way.

generic attribute, say 'average weight over some time period', we should be able to both specify the overall period, and specific points within that period where 'measurements' are to be taken as well as, of course the ecological entity of interest. So, when the simulation model is run, a value for the daily average weight of a particular sheep for a particular day with say 24 hourly measurements can be computed. It may do the same for a different day. It may require the maximum daily average over the course of some week; or possibly the maximum daily average for a particular day but over a set of sheep. Our formalism supports this kind of variety in a principled uniform manner.

The second case where time is relevant is with regard to processes themselves. The time scale over which the effects of different processes are noticeable vary greatly. Lions preying on wildebeest is relevant over a period of hours, but it would make no sense to perform computations over seconds. Conversely, processes like precipitation which may transport minerals etc happens continuously. The ability to associate time scales with processes gives a limited ability of the system to perform consistency checks. Also, ideally the simulation model should be able to use appropriate nesting to account for processes occurring on different time scales. This is a compiling issue for which there may well be standard techniques available. We have not looked into this. In this thesis, we only discuss examples where there is a single uniform time scale, as in our Serengeti example.

On the whole, our requirements for representing time are not excessive. The facilities we provide are thus unsophisticated, although there is considerable scope for enhancing them, especially where nesting is required in final programs to accommodate processes occurring on different time scales. Further discussion of time is deferred until § 5.6.6.2.

2.6 Summary and Conclusion

We have explored the domain of ecological modelling. We have identified a fundamental distinction between modelling information and ecological information. At the modelling level, we identified what is required to represent simulation models. At the ecological level, we identified what is required to describe ecological systems. It is also important to represent explicit links between these two levels to facilitate model comprehension.

Most representations of simulation models contain no explicit domain information at all. This is the root cause of the difficulties associated with the two primary issues addressed in this thesis: model comprehension and model construction. Our basic approach to facilitating model comprehension is to enable modellers to explicitly relate a description of an ecological system with a simulation model of it. Our basic approach to assisting in model construction is to (a) allow modellers to 'speak' in ecological terms (thus reducing conceptual distance) and (b) identify and constrain the modelling search space. The approaches to both model comprehension and model construction rely fundamentally on (a) representing domain information, and (b) making a formal connection between the ecological concepts and their idealised representation in the simulation model.

We now summarise the important relationships between ecological and modelling concepts that we have discussed or hinted at so far. First, there is the idealisation of the real world which is manifest in our conceptual modelling framework.

1. most 'things' in an ecological system are idealised in terms of processes, entities, attributes, and/or values.
   e.g. substances are idealised as attributes denoting the amount of the substance in some entity.
2. an entity with substructure is idealised as a set of [component] entities; the whole may be a set or a composite entity
   e.g. a population is idealised as a set of like entities
3. a process is idealised as a collection of effects each of which changes the value of some ecological attribute in some way.

Second there is the idealisation of the conceptual model [of a particular ecological system] which is manifest in the simulation model.

1. the ecological system is idealised as the simulation model
2. an entity is [implicitly] idealised as a set of ecological model variables; entities need not be explicitly represented in the simulation model
3. an ecological attribute is idealised as an ecological model variable
4. an effect [of a process] is idealised as an effect variable, (usually a partial rate variable) in conjunction with some mathematical relationship for computing it.

5. an influence is idealised as a [computational] dependency between two or more model variables.

The mechanisms for achieving these explicit relationships between ecological and modelling concepts are described in chapter 5. We noted two key reasons for distinguishing between the ecological and modelling levels.

1. to ensure model comprehension
   i.e. that important model assumptions and decisions are represented.
2. to facilitate the identification of the modelling/idealisation search space

We stressed the importance of the former, the latter is discussed in more detail in chapter 4. In identifying what about ecological systems and models needs represented, we have considered most of what is necessary regarding the expressive power issue. This in turn serves to identify the ecological modelling search space. What we have *not* yet done is:

1. identified the range of information required to bridge the gap between ecological system descriptions and simulation models. This is the conceptual distance issue.
2. considered how to control search by pruning inappropriate options, or advising on good options. This is the choice management issue.
3. considered the issue of syntactic adequacy; this manifests itself in the design of the user interface of ELK.

The issues of conceptual distance and choice management are dealt with in chapters 3 and 4. Syntactic adequacy is discussed in chapter 7.

Chapter 3

Ecological Modelling Goals

3.1 Introduction

So far, we have considered what information needs to be represented to describe ecological systems and models, and indicated how this helps achieve model comprehension. We have not considered the process of model construction.

Put in terms of the major sources of difficulty of formalisation problems, we have only been concerned with expressive power issues. We must as well explore the issues of conceptual distance and choice management. We must identify the requirements of a computer assistant for eliciting formal descriptions using these formalisms. This chapter is devoted to dealing with one important aspect of these requirements: ecological modelling goals. Chapter 4, deals with other important aspects. Goals play an important role in bridging the ecological and modelling levels of information. They also play a role in controlling choices.

The primary objective of this chapter is to address our goals objective (see page 4) in the context of ecological modelling. We explore the nature of ecological modelling goals and consider how they may be used to guide the formalisation of ecological systems. We begin with a brief discussion of the modelling process in general, and the role of goals. We describe a survey of the ecological modelling literature (from the perspective of goals) and the resulting classification of goal types and uses. We show how this classification might be used as the basis of a control strategy for a model acquisition system. We outline the requirements for representing a key subset of these goals. The formal details are deferred to chapter 5.

That goals are useful in ecological modelling seems to be widely accepted. When general methodology is discussed in the ecological modelling literature, goals and/or objectives are frequently given a place of high importance. The general consensus is that one way to assist in the construction of ecological models is to consider *first* what the purpose of the modelling exercise is. For example:

> "... the appropriate modelling technique for ecosystem analysis depends on the object to which the model is to be applied."[Bledsoe, 1976]

As well as being useful for constructing models, goals are crucial to understanding and using them. This alternate view is expressed in [Rothenburg, 1989]:

> "It is widely recognised that the purpose of a model must be understood before a model can be discussed. [p 76]"

Unfortunately, in the various accounts where such claims for the usefulness of goals are made, very few details are ever given. No specific examples are cited to substantiate the claims; nor is any general advice offered.

To meet our goals objective, we set out to substantiate these claims and embody them in a computer assistant. The idea is that it should first acquire from the user a statement of their goals, and then guided by this [and other] information, acquire a precise description of the model. Before we could begin building such an assistant, we first had to:

1. identify and classify ecological modelling goals (§ 3.2).
2. find out how these goals may be used (§ 3.3).

Although the results of this analysis have been illuminating, and greatly influenced the course of this research, much of what is discussed in this chapter is not directly incorporated into the current implementation. Nevertheless, we do elaborate on the role of this analysis in this research, why the current implementation does not include it, and how it may be more directly incorporated in future versions of ELK.

3.2 A Goal Survey

We examined several pieces of literature on general modelling methodologies and a few dozen descriptions of specific models. For each, we identified and classified the goals that were explicitly mentioned. If none were mentioned, we made educated guesses as to what the goals probably were, or might have been. This yielded several dozen goals which were then classified.

Concurrently, we wrote down anything that was mentioned about how goals were used. We were most concerned with goals that provided a focus for the modelling exercise, and therefore might have been used to help construct the model, but other uses were also noted. Educated guessing was again required. The results of the analysis of how goals may be used is presented in § 3.3.

The literature on general modelling methodologies often contained very general classifications of types of goals. For instance, the following [abbreviated] list is taken from [Berman, 1979].

- to describe data
- to test a hypothesis
- to formalise concepts
- prediction

This serves well as a starting point, however, we needed specific examples as well. To find these, we consulted a selection of published models. Many were obtained by browsing through issues of the journal, Ecological Modelling. Very occasionally, the authors explicitly state their general goals and/or specific objectives of the modelling exercise. Most of the time, however, we were forced to infer what the objectives actually were, or might have been. This was done by examining the sections of a paper where the results are described (*e.g.* discussion, conclusion). For example, an author may conclude (among other things) that: "an increase in the minimum annual water temperature results in a catastrophic decline in the population of jellyfish"[Legovic, 1987]. This gives rise to a number of educated guesses of goals that could have been used to guide the construction of that model. These include:

- What affects the size of the jellyfish population?
- What is the effect of temperature on jellyfish?

- What caused the catastrophic decline in the population of jellyfish?

3.2.1 An Ontology of Goals

An ecological modelling goal is defined to be:

> *A reason for engaging in the exercise of constructing a model of an ecological system.*

We are deliberately vague about the type of model because some of the goals may be achieved strictly by building conceptual models without computer implementation. In figure 3–1 we present a goal ontology which constitutes a classification of the types of goals that were found in the literature. We begin with very general goal categories and proceed in one of two ways:

1. we identify different examples or kinds of the general category thus defining sub-categories of goals, or goal *subtypes*.
2. we identify specific categories of goals which *facilitate* the achievement of goals in the general category; thus defining categories of *sub-goals*, or *sub-tasks*. By 'facilitate', we mean the subgoal is either necessarily required for or serves to enable the achievement of the goals in the more general category.

This defines a graph. There is a single node type; these denote classes or categories of goals. There are two kinds of arcs; these correspond to the above two cases (*subtype*, *facilitate*). It is not a tree as a few goal types appear more than once (*e.g.* [3] ≡ [2(b)iA]. Figure 3–2 gives the graphical version of figure 3–1.

There is a large amount of diversity in the sorts of things being said. In this section, we introduce and discuss the categories in this ontology and their interrelationships. We consider three major categories for ecological goals (from figure 3–1):

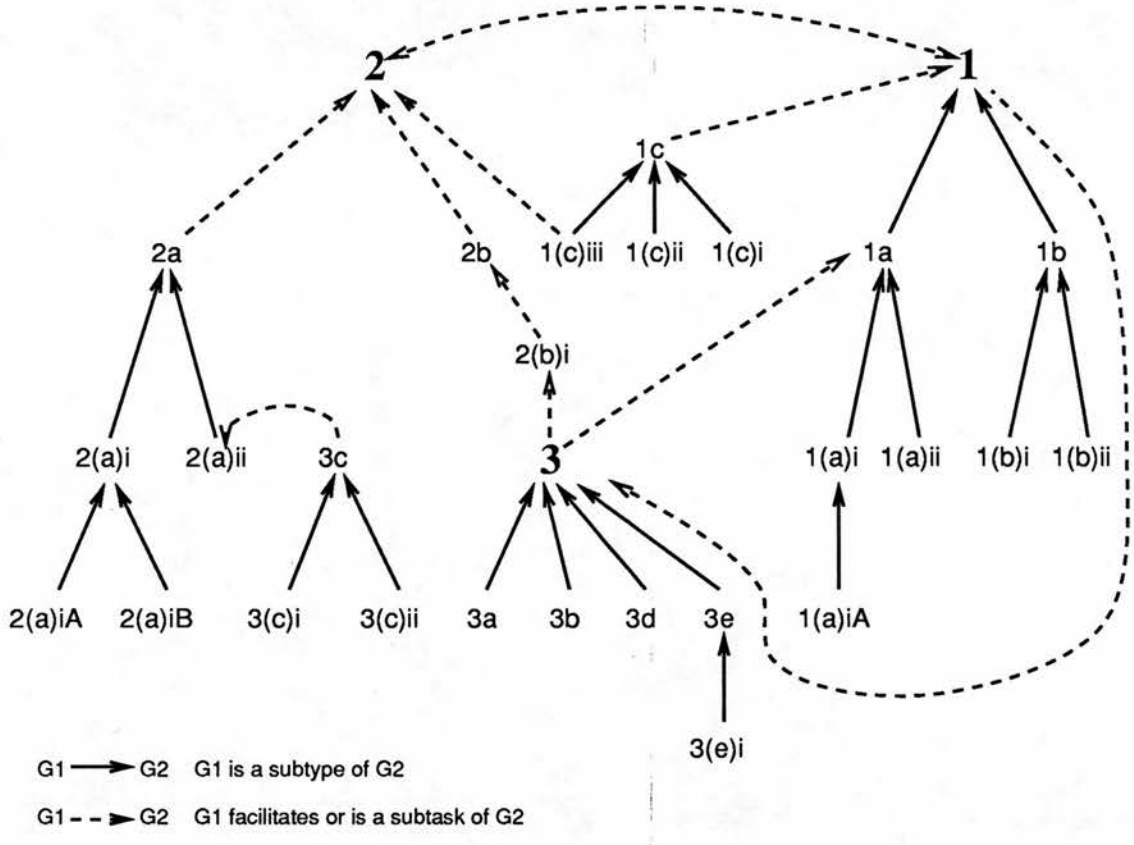
1. To have a model which represents to some acceptable level of accuracy some aspects of an ecological system.
2. To enhance understanding of some real world system
3. To answer some specific question about an ecological system, or a model of same.

1. *To have a model which represents to some acceptable level of accuracy some aspects of an ecological system.*
   (a) for interrogation
      i. for prediction (quantitative)
         A. for management
      ii. for qualitative behaviour
   (b) to match a set of data
      i. empirical model
      ii. mechanistic model
   (c) ⋆ *model-building subgoals*
      i. to find the value for some parameter(s) for a model
      ii. to lay the foundation of ideas and methods for other models
      iii. ⋆ to formalise concepts regarding some aspects of an ecological system
2. *To enhance understanding of some real world system.*
   (a) ⋆ to test current understanding
      i. to identify topics for further research
         A. Gaps in field data
         B. Gaps in understanding of ecological system
      ii. to test a general theory
         A. ⋆ to test a hypothesis [≡ 3c; details below]
   (b) ⋆ To extend current understanding.
      i. ⋆ To explore the relationship between two or more aspects of an ecosystem
         A. ⋆ To answer some specific question about an ecological system. [≡ 3; details below]
   (c) ⋆ to formalise concepts regarding some aspects of an ecological system [≡ 1(c)iii above]
3. *To answer some specific question about an ecological system or a model of same.*
   (a) What causes a certain system behaviour?
   (b) How can a certain system behaviour be achieved (possibly given some constraints)
   (c) To test a hypothesis
      i. Is it the case that the affect of X on Y is Z?
      ii. Does X cause Z?
   (d) What is the affect of X on Y?
   (e) What outputs can I expect given these inputs?
      i. To plot Y versus X
         - plot Y versus time
         - what is the value of Y at time T?
   (f) ⋆ To have a model which represents to some acceptable level of accuracy some aspects of an ecological system. [≡ 1 above]

By default, the item-subitem pair denotes the 'a kind of'/example/subtype relationship.

'⋆' denotes facilitation/subgoal/subtask relationship.

Figure 3–1: Goal Ontology



This is graphical depiction of figure 3–1. One important relationship shown here was omitted in figure 3–1: the major categories 1 and 2 are mutually interdependent; each facilitates the other. This reflects the fact that the process of constructing a model helps increase understanding, which in turn helps to enable a better model to be constructed. Note also the link between 3 and 1a.

Figure 3–2: Goal Graph

3.2.1.1 To have a model ...

We begin with category 1. This is a generic goal which is at the heart of any modelling exercise. An example from [Holt et al, 1987] is:

> "... to formulate a descriptive model to simulate the role of predation in the life system of leafhoppers in paddy fields based on field data."

Models are constructed with different purposes in mind. Some models are meant for general interrogation [1a]. Various questions will be posed about specific elements and relationships in the model. A variety of simulation experiments will be run and the resulting behaviour analysed. The nature of the questions is discussed later (see [3]). One possible use of this sort of model is for prediction [1(a)i]. If this is the case, the model will require careful validation. A model which is useful for prediction may in turn be used for management purposes [1(a)iA]. This may be the primary goal of building a model. An alternative to a model for prediction, is a model that is only required to be able to replicate the general qualitative behaviour [1(a)ii] of a system.

A second purpose for building a model is to match a set of data [1b]. In this case, the end model itself need not be intended for actual use. If the primary goal is merely to match the data, then the exercise is an empirical one[1(b)i]. Alternatively, it may be a requirement to model certain mechanisms in the ecological system explicitly[1(b)ii]. As an aside, many would argue that the usefulness of the purely empirical exercise just mentioned is very limited.

The last sub-category we consider for the first major category is: *model–building subgoals.* Each goal in this class can be viewed as a subgoal of the first type of goal. That is, model building subgoals are things that are required for the construction of the model [by definition!]. The diversity of such things is great, and their identification constitutes defining much of the simulation modelling search space. Here we cite only examples from our survey where the stated reason for building a model is really a task required for the construction of some other model. We found three examples of this type:

- to find the value for some parameter(s) for a model [1(c)i]
- to lay the foundation of ideas and methods for other models [1(c)ii]
- to formalise concepts regarding some aspects of an ecological system [1(c)iii]

In the case of determining a parameter, it must be assumed that a substantial part of a model already exists. It is being fine tuned. The existing model may or may not be a computer model. There are two senses in which a model may be constructed for another model. For one, large complex models are often built up of smaller submodels. In some cases, these submodels are themselves quite substantial. Also, some detailed (low-level) models are constructed for the purpose of enabling a simpler model to be used at a higher level. The more detailed one serves to validate the higher level one.[1] Finally, formalising some ecological concept is a necessary part of any ecological modelling exercise. To have this as a primary goal suggests that the modeller is more interested in resolving theoretical issues than in using a model after it is constructed.

3.2.1.2 To enhance understanding ...

This naturally leads us to the next major category of goals. It is fairly common for authors of modelling papers to explicitly state that one of their objectives in engaging in the modelling exercise is *to enhance understanding of some real world system*[2]. The very process of constructing a model forces a researcher to face up to a myriad of issues some of which may not even have been thought of yet. As in the last example in the previous paragraph, to have the model itself is not the main motivation. Rather, it is the process of constructing it which forces a greater understanding of the ecological system. There are various ways to achieve this. We identify three sub-categories of goals which facilitate the goals in this general category.

- to test current understanding [2a]
  - to identify topics for further research [2(a)i]
  - To test a general theory [2(a)ii]
- to extend current understanding [2b]
- to formalise concepts regarding some aspects of an ecological system [1(c)iii]

Identifying areas for further research may take the form of finding out what data still needs to be collected[2(a)iA]. Alternatively, it may consist of identify-

---

[1] This last example was not found in the literature by the author, rather it was discovered by personal communication with Robert Muetzelfeldt.

ing specific gaps in understanding of one or more aspects of the ecological system of interest[2(a)iB]. Secondly, current understanding may be tested by subjecting a general theory to a test. We can view a theory in this context as a very general sort of hypothesis, which comprises a coherent set of possibly many sub–hypotheses involving certain aspects of the ecological system. There are many types of hypotheses that may be tested. These take the form of specific questions; they are dealt with when we consider the last major goal category. This completes the discussion of testing current understanding.

We now consider goals about extending current understanding. By this we mean the identification and/or explanation of mechanisms which are responsible for some observed behaviour in an ecological system. One means of extending current understanding (and a common goal in its own right) is: *to explore the relationship between two or more aspects of an ecosystem*[2(b)i]. For example, "*To explore the relationship between the populations of jellyfish and their prey*". This is still extremely general. Making this specific entails asking specific questions, such as "*What is the affect of removing one of the jellyfish prey populations on the jellyfish population itself?*" . The nature and diversity of the questions that may be asked is discussed next. Note that to answer a specific question is not an example/subtype of exploring a relationship between two or more aspects of an ecological system; rather it is a means to that end. Lastly, we note that to formalise concepts about some ecological system is also a vehicle to enhancing understanding. This was also a model–building subgoal. Since this is really the heart of the matter, it is not surprising that it occurs twice.

3.2.1.3 To answer specific questions ...

We now turn to an analysis of various types of questions that may be asked of a model and/or of an ecological system. In the previous two major categories, the goals have been fairly general; now we get more specific. Not surprisingly, there is a tremendous variety of questions that one might ask. Furthermore, most of the specific questions that we will discuss apply to both of the other major categories (*i.e.* the major categories are not mutually exclusive). Note that although we explicitly give an analysis of *question* types, keep in mind that we are implicitly categorising *goals*. Each question has a corresponding goal: *To find an answer to the question.* It is simply a matter of grammatic convenience to talk about

questions rather than goals. All the examples that we came across in the literature could be grouped into one of the five basic types:

1. What is the affect of X on Y? [3d]
2. What outputs can I expect given these inputs? [3e]
3. To test a hypothesis [3c] (and [2(a)i])
4. How can a certain system behaviour be achieved (possibly given some constraints)? [3b]
5. What causes a certain system behaviour? [3a]

The first type is a very general sort of question. We gave one example in § 3.2.1.2 (X was 'removing one of the jellyfish prey populations'). Another example is: "*What is the affect of an increase in minimum annual water temperature on the population of jellyfish?*". For now, simply regard X and Y as arbitrary ecological entities, concepts or subsystems. We shall consider the nature of X and Y in § 3.4. The second type (about inputs and outputs) is also quite general, and very common. A more specific subtype of this is: *To plot Y versus X*, which is also frequently quoted as a goal. An interesting case is when X is *time*. This is an implicit goal that creating any output variable gives rise to. Even more specific than this is a goal to know the value of some variable at some specific time. *These latter goals are of fundamental importance.* They underlie any use of any quantitative simulation model. Furthermore, they can be used to drive an entire model elicitation process as in [Robertson et al, 1987; Haggith, 1990]

The third type of question is to test a hypothesis. A general example is: *Is it the case that the affect of X on Y is Z*, where *X* and *Y* are as described above, and *Z* is a description of some arbitrarily complex condition or state of affairs involving X and Y (*e.g.* some relationship between X and Y). Another very general example of testing a hypothesis would be: *Does X cause Z?*. For a particular ecosystem, this might instantiate to "*Does an increase in the minimum annual water temperature cause a catastrophic decline in the population of jellyfish?*". Note that the space of hypotheses is virtually infinite.

A fourth type of question is: *How can a certain system behaviour be achieved?*. Two examples of this are: "*To limit the deer population to some fixed amount, given certain available resources*" and "*To maximise profits at a fishery*". This is just the sort of question that is put to a model being used for management purposes. The last question we consider is: *What causes a certain system behaviour?*.

This might instantiate to: "*What causes a catastrophic decline in the population of jellyfish?*". This is similar to the previous one. They differ in emphasis. Here, we are most concerned with understanding what causes an observed system behaviour. For the previous question, the end result may or may not be well defined. Furthermore, the aim is not better understanding, but rather to achieve a certain result.

Finally, there is one important facilitation relationship between this and the first major category. Specifically: to have a model which represents to some acceptable level of accuracy some aspects of an ecological system serves to enable the asking and answering of questions such as *What outputs can I expect given these inputs?* or *What is the affect of X on Y?* (see dashed link between **1** and **3** in figure 3–2). This completes the presentation of the basic content and structure of the ontology for ecological modelling goals. Next, we consider how these goals may be used.

3.3 Using Goals

The identification and classification of goals is just the first step. By itself, the resulting ontology is of minimal interest. However it serves as a basis of an analysis of how and whether goals may be used in the process of ecological modelling. As noted in § 3.1 much of this is not incorporated in the current version of ELK. In this chapter only, we always use future referents when describing uses for goals even though some of this is implemented; normally we will only use future referents for work not done. At the end of the chapter, we indicate the state of the implementation with respect to goals.

We considered each goal in the ontology, and determined whether and how it might be useful. These uses were then classified. The details of the uses for each goal in the ontology are presented below; first we summarise the main findings. How a goal can be used depends fundamentally on the nature of the information that it contains, as well as the details of that specific information. With respect to the nature of the information that a goal contains, we distinguish between goals that mention specific concepts about the ecological system or model (*e.g.* entities, attributes, variables, processes, outputs) and those that do not. We refer to goals that do as *low-level* goals. For example, the low level goal 'how does temperature

affect jellyfish?', mentions temperature and jellyfish. Goals that do not refer to specific concepts about the ecological system or model are referred to as *high-level* goals (*e.g.* 'to have a predictive model').

3.3.1 Kinds of Goal Uses

We identify two uses for each of low- and high-level goals. This is discussed in the context of a modelling consultation consisting of an ecologist who wishes to construct a model, and an expert modelling consultant (human or computer).

First, and most importantly, low-level goals indicate that certain things are important to the person constructing the model (*e.g.* temperature, jellyfish). This information may be used by the consultant to focus the discussion on the concepts that are mentioned. This is a form of choice management; in particular it helps the user decide *what* to do next (*i.e.* dialogue control).

The second possible use for low-level goals is to help define simulation experiments. Suppose the goal is to find the temperature which results in the maximum yield in a fishery. This suggests that the model will need to be run several times with different values for temperature, and results compared. This constitutes a simulation experiment which the computer assistant might automatically define, given this goal and other information.

The first use for high-level goals is to serve as starting point for eliciting other more specific goals. For example, the high-level goal 'to have a predictive model' may prompt the expert to ask what it is that the model needs to predict. This might result in the low-level goal 'to predict the size of the jellyfish population' which may be used as described above. This use of high-level goals is similar to the first use of low-level goals in that the role is to direct the nature of questions asked during a consultation, however it is less direct. Instead of focusing the discussion on specific concepts, a high-level goal may be used to focus the discussion on acquiring other more specific goals eventually leading to low-level goals. These low level goals in turn may be used to focus the discussion on specific concepts.

Secondly, high-level goals may be used in conjunction with modelling expertise to constrain modelling decisions. For instance, given the same ecological system to be modelled, certain decisions will be made differently depending on whether the end model is to be used for prediction. The precise details of how this might affect decisions are dependent on many factors. We defer discussion of this matter

to § 3.3.3. This second use of high level goals is also a form of choice management; in particular, it helps the ecologist decide *how* to do something.

Summarising, we have identified the following uses for goals:

1. to focus the dialogue between the ecologist and the consultant
   (a) high-level goals may be used to direct the elicitation of more specific goals
   (b) low-level goals may be used to identify specific ecological and/or modelling concepts of importance about which further questions can be asked.
2. to help suggest an appropriate modelling method or decision (for both high- and/or low-level goals)
3. to help define simulation experiments.

It is useful to distinguish *when* in the course of a modelling exercise, these goals may come into play. Uses 1a and 1b are relevant in the early stages of model elicitation. Use 2 is relevant in the middle and later stages. Use 3 is relevant only after the model is fully specified (or nearly).

3.3.2 Using Different Kinds of Goals

We now consider how each of the various goals in our ontology may be used. We consider the main categories in order. First, we consider the goal: *To have a model which represents to some acceptable level of accuracy some aspects of an ecological system*[1]. In this form, it is far too general to be of much use other than to elicit more specific goals [use 1a]. There are two dimensions of variation implicit in this goal. One is the degree of model accuracy required; the other is the aspects of the ecological system (*e.g.* the relationships between the jellyfish population and its prey populations). We consider each dimension separately. To have named specific aspects of the model is to provide hooks for asking further information [use 1b]. Knowledge of the degree of accuracy required can be used to help decide on certain modelling choices. For example, a general goal of the form "*to be used for prediction and management*" suggests that there should be sufficient data to guarantee reliability. This places constraints on the nature and extent of idealisation that the model may embody [use 2].

Next we consider *model–building subgoals*, an important sub-category in the first main category. The first goal in this (sub)category is to find a value for a parameter. This goal assumes that there is already a model in existence. If the model is a computer model, then this goal may be used to define simulation experiments to determine the parameter value given other constraints[use 3]. If it is a theoretical model, then mention of a parameter can spawn questions about what specific concepts the parameter is relevant to [use 1b]. The second goal in this category is *to lay the foundation for other models.* This really just begs the question: "Why is the other model being constructed?". Otherwise, this goal is of little use. Finally, in this category we have the goal *to formalise ecological concepts.* Use 1b is relevant here also. The specific concepts that are mentioned can spawn further questions about them. Another possibility is for the system to ask the user what the goals are for wanting to formalise the concepts. That is, what are the criteria for success. If it is for theoretical interest only, then there is not much more to do. If, on the other hand, it is for the purpose of constructing some other model, then the user may be asked to state their goals for that larger enterprise [use 1a].

We now consider the second general goal category: *to enhance understanding of some real world system.* As for the first example, this goal by itself is only useful for eliciting more specific goals [use 1a]. However, if specific aspects of the system are mentioned, then goal use 1b applies. We now consider the three goals under the subcategory *to test current understanding.* The goal *to identify further research topics* is of no obvious use. The goal *to test a hypothesis*, can be used to provide hooks for asking further questions; but only if the hypothesis is sufficiently specific to mention certain ecological entities and/or concepts about which further questions may be asked[use 1b]. Similar comments apply to the goal *to test a general theory.* This is not surprising, as a theory is really just a big hypothesis. The user would have to indicate what the theory relates to. This can be used to ask further questions. The next subcategory is *to extend current understanding.* Such a goal by itself is of no use; again, the user would have to say what it is about the ecological system that they need to know more about. This would provide hooks for asking more questions[use 1b, yet again].

The final main goal category is *To answer some specific question about an ecological system or a model of same.* We have seen that there are a large number of

questions that one might ask. The more specific the question, the more useful it can be in the early stages of model elicitation. This is use 1b above. As we saw in use 2, some very specific questions can also be used to guide decisions about how do do something. These will usually mention particular entities and/or concepts in the ecosystem. Each such concept that is mentioned will usually require further elaboration. For instance, if a modeller mentions a jellyfish population in a goal, then the system should ask what attribute(s) of the jellyfish population are important. Another kind of elaboration consists of specifying a chain of causal relationships. For instance, if the goal is: "*What is the affect of temperature on jellyfish?*" the system might ask if the temperature affects the jellyfish directly, or whether there is some intermediate factor. Temperature could affect food supply which in turn affects the jellyfish. A specific goal like

> "*to test the affect of changing the minimum annual water temperature from 8 degrees to 4 degrees on the size of the jellyfish population*"

has implications about a specific simulation experiment that needs to be performed on the eventual model. As such it would be useful after a model was already constructed. However, knowing that such questions were going to be asked of the eventual model could guide the model construction process. This goal suggests that the jellyfish population size needs to be a model output. This means, that it is not a parameter. Thus, it is possible to use the goal to help decide how a certain ecological entity is to be represented.

3.3.3 Conclusion: Using Goals

The main conclusions that we can draw from this goal use analysis are:

In the early stages of model elicitation:

- High level goals may be used only as a way to coax the ecologist into expressing more specific low-level goals. The high-level goals are not useful directly.
- Low level goals may be used to identify the ecological and modelling concepts that are important. This may be used to focus the ecologist's attention on elaborating specific parts of the ecological system or model.

In the middle and later stages of model elicitation: Both high and low-level goals may be used to guide modelling decisions, and to define simulation experiments.

We have chosen to concentrate our efforts on the early stages of model elicitation. This is for three reasons. First, because we are interested in a system that can build models from scratch, it is a natural place to start. Second, we are not concerned with defining simulation experiments. Third, and importantly, there is a significant barrier to providing assistance with respect to using goals to guide modelling decisions. Namely, it requires a body of knowledge that relates goals to modelling decisions. Unfortunately, this knowledge is not easy to come by. It is not found in books, and it is not easy to extract from expert modellers. Thus, an extensive knowledge acquisition exercise would be necessary and there would be no guarantee of producing a useful corpus of knowledge. Instead, we recognise this as an important problem for future research.

Because the first use of high-level goals is indirect (to coax the user into giving low-level goals), and the second use is relegated to future efforts, this thesis is directly concerned only with low-level goals. We wish to identify a fundamental subset of low-level goals, which will form the core of any future version of ELK which also used high-level goals. A key observation is that ultimately, the only way to achieve any modelling goal (high- or low-level) is by inspecting and/or interpreting tables and graphs produced by the model. We argue, therefore, that the fundamental subset consists of those goals cast in terms of X and Y (*e.g.* 3(c)i, 3d, 3(e)i).

3.3.3.1 A Goal-Driven Dialogue Graph

We proceed by showing how the goal ontology could be used as a dialogue graph in a future version of ELK, and argue that goals cast in terms of X and Y are of chief importance. Each node on this graph corresponds to a goal or goal category. When a specific goal or goal category is being considered, we say that the corresponding node is 'active'; *i.e.* the dialogue is now centred on eliciting a goal from that category. We argue that no matter what node in this graph a user may begin at, they will necessarily be led to nodes corresponding to low-level goals cast in terms of X and Y. This is a direct consequence of our observation that the initial use of high-level goals is to coax the ecologist into expressing low-level goals. Thus

in dialogue terms, if a user begins with a high-level goal, they are quickly led to low-level goals. Any high-level goals that may have been specified along the way, would not be used again until later. If a user begins with a low-level goal, the consultant may immediately make use of the specific concepts mentioned. There would be no need for any high-level goals except perhaps in the middle or later stages in the model specification.

To elaborate this argument we give some details and examples of how the goal ontology in figure 3–2 may double as a dialogue graph. We do so by describing various hypothetical scenarios. The system could initially present the user with the three high-level options corresponding to the three major goal categories. From there, the system may encourage users to be more specific in one of two ways. First, the class may be specialised by choosing one of the subclasses connected by *subtype* arcs. Alternatively, one of the goal classes connected by a a *subtask* arc may be chosen; this amounts to backward chaining.

If the user chooses the first category, (To have a model... [1]), they would then see a menu of three choices (1a,1b, and 1c). Suppose they chose [1a] (interrogation). The next menu would offer choices between either of the two kinds of interrogative goals (quantitative prediction, or qualitative behaviour). Additionally, the menu would include the goal category "To answer questions..." [3] which is a subtask rather than subtype link. Suppose prediction was chosen. This is still a high level goal which at this point can only be used to guide the user to a more specific goal. Choice [3] is still relevant; suppose the user took this option next. The system can then follow the subtask and subtype arcs corresponding to each kind of question in this third major category. A user might decide to specify a goal of the sort "*What is the affect of X on Y?*". These are still high-level goals. The system then may encourage the user to specify exactly what X and Y are (*e.g.* temperature and jellyfish). This then provides a useful starting point to the whole model elicitation process.

If earlier, the user had selected the option to match a set of data [1b] (rather than [1a]), this begs the question "What data?". This then leads to questions about relevant ecological entities in much the same way as the affects goal does. As noted above, the high-level goals specified along the way will be useful much later when it comes time to make certain modelling decisions.

Alternatively, if the user started right at the beginning with the second major

category, (to enhance understanding), rather than the first, this quickly leads via a different route to the elicitation of specific information about the ecological system or model components. For instance, if they want to identify gaps [2(a)i], the system asks what they know already. If they want to test a general theory [2(a)ii], then the system could ask what specific hypotheses they wish to explore. This leads to questions about ecological entities and processes etc.

If the user initially selected the third major category (to answer questions...), this is a still quicker route to talking about specific ecological and/or modelling concepts. Later during the modelling process, users would always have the option to express other higher level goals. Or in the course of designing the model, the system might prompt for other goals. For instance, when trying to make a decision, the system might backward chain on a rule like: "If the user wants a prediction model, then ...".

This illustrates our point. None of the high-level goals are of any immediate use in the model elicitation process. In order to make any real headway we need to get down to the level of specific ecological and modelling concepts. The most immediate route to getting specific ecological concepts using goals is via the third goal category. That this was the case emerged from our analysis of goal use in § 3.3. By far the predominant use was to identify items of importance in the ecological system or model about which further questions can be asked. This helps the user decide *what* to do. In order to realise this, it is necessary for goals to refer to specific things in the ecological system or model. As mentioned in § 2.4.1.2 (p 50) a very simple goal of the form "What is the value of Y at time T" may be used to drive the model elicitation process. Such goals define output variables.

In the next section we discuss the nature of the ecological and modelling concepts that are contained in low-level goals. To a large extent we have already covered this in chapter 2; there are a few additional things.

3.4 Representing Goals: Requirements

Due to the above conclusions, we make no attempt to formalise the whole ontology just described. Instead we deal only with the lower level goals. These are the most useful in the first instance. We recognise that it might be useful for certain ecologist users to express their goals initially at the high level as a way to gently lead them to more specific and more useful goals. However, since the high-level goals in and of themselves are not directly useful, we have deferred that task.

Of the low-level goals, we consider only the simplest most directly useful examples. In the specific questions category, the first two (3(c)i and 3(c)ii, figure 3–1) require representation of some kind of system behaviour. This involves a possibly complex characterisation of interrelated entities, attributes, and processes. Furthermore, to achieve such a goal entails a potentially highly complex problem solving activity involving planning, scheduling, or possibly optimisation. Similarly, the Z part of the hypothesis questions is quite open ended. This is not the appropriate place to start, we need smaller building blocks. We begin therefore with the most basic questions:

- What is the affect of X on Y?
- To plot Y versus X?
  - plot Y versus time
  - what is the value of Y at time T?

Specific instances of goals of these types give a good start in identifying what the most important entities and processes in the ecological system are and how they depend on each other. These in turn give rise to model variables and equations.

We first consider the simpler of the two goal types: the plotting of two variables. The representation requirements here are very straightforward. Model variables are all that we need. Model variables and their relationship to attributes have been discussed extensively in chapter 2; the formal representation details are given in chapter 5. Note that to have a goal of the type: 'to plot Y at time T' is exactly equivalent to saying that Y is an output variable.

We note some restrictions: for instance, it makes little sense for the dependent variable Y to be an exogenous variable because they do not not depend on anything in the model. Suppose temperature was an exogenous variable. The only thing

that it makes sense to plot temperature against is time. But, this is a model *input*, not an output. The independent variable, X will normally be a proper model variable. If it is a parameter, this has special implications. Specifically, it requires running the simulation several times with different values each time. This kind of experiment is common.

The *X affects Y* goal is very similar to *plot Y versus X* with some additional complexity. To ask how X affects Y is really to suppose that the specification for X entails a description of some change (we use C_x to denote this). For instance, the temperature may 'increase'. Something may be 'included or not' in the model. The change must be something that is in the direct control of the modeller. For instance, it is not sensible to ask what the affect of an increase in temperature is on the average size of some population if the size of the population was modelled as a state variable. Except indirectly, the modeller cannot control what the average value of a state variable is. It is thus not useful to ask a question about how altering it affects something else. It is of course possible to *plot* a state variable against anything or vice versa. The list of options for C_x that we identified during our survey is given below:

- change the values (of some attribute or variable). Exactly *how* the values change can be specified in various ways
  - increase/decrease
  - different ranges of values
  - sensitivity analysis
- change in the method of computing the attribute (*e.g.* one equation instead of another.)
- Something may be included in the model or not. This can be manifest in various ways depending on the nature and complexity of the 'something'.

With respect to what X and Y can be, the discussion of attributes in chapter 2 covers most of what we need. Some relevant concepts gleaned from the goal literature not yet discussed are:

- equilibrium; various aspects of the equilibrium include:
  - actual level
  - stability
- oscillatory behaviour; aspects of this include:

- period of oscillation
- amplitude of oscillation

- slope; *i.e.* is it increasing/decreasing or constant)
- threshold value with respect to some other variable; aspects of this include:
  - existence of a threshold value (*i.e.* yes or no)
  - actual threshold value

The notion of stability needs to be strictly defined in order for the system to be able to detect its presence or absence. For this, special purpose machinery could be added. Related to this is the notion of oscillatory behaviour. The period and amplitude could be represented as simple attribute names (*e.g. osc_per*, and *osc_amp*) but this misses the meaning of the concept which in theory could be ascertained by the system monitoring the values of various variables under certain conditions. Again, special purpose machinery would have to be invoked.

Many of these can be viewed as induced attributes similar to average, maximum etc. That is, an attribute like population size, which has values over some range of times, induces the notion of an equilibrium which may or may not exist. The attribute 'equilibrium level for population size' is defined in terms of the size attribute in a way analogous to the attribute 'average population size'. A difference is in how the value of the induced attribute is obtained. For average, etc, we have a relatively straightforward computation where all the inputs are readily available. In exactly the same way that we need a separate algorithm for computing average, we need separate machinery for determining such things as threshold, measures of stability, etc. For most of these, it entails monitoring the ecological system as a whole (or the simulation, in modeling terms). So, where for average we need only the set, and a function to compute a value, for equilibrium, threshold etc., we need a potentially large chunk of the model complete with functions, variables, initial values, a time structure, etc. These induced attributes [*e.g.* equilibrium value] apply to the ecological system as a whole, not to specific entities. The model variables that they give rise to describe the simulation model. This contrasts with the situation for average, say, where the induced attribute 'average weight' applies either to a sheep or a set of sheep. These attributes are special in another sense as well. They *may not* be used as dimensions for defining substructure. This is largely because they are not really attributes of entities within the ecological

system, but of the ecological system itself. We give the formal details of how to represent equilibrium and threshold in § 6.5.1.1.

3.5 Conclusion

In this chapter, we have explored how goals may be exploited in the process of constructing ecological models. We described the results of a survey of ecological modelling literature from the perspective of goals. We identified and classified the types and uses for goals. This resulted in an ontology for goals which may serve as a basis for a goal–directed strategy for the early stages of model elicitation. Finally, we summarised the requirements for representing a subset of the goals in our classification. The conclusions that we draw from this chapter are:

Goal Ontology

- There are many kinds of ecological goals. Some are very general, others very specific. Some address issues at the modelling level, others at the ecological level. For example, some entail the model being of primary concern; for others, the model is a means to some other end.
- We do not make any strong claims that our ontology covers every possible goal. However, we feel confident that it covers a large fraction. There have been several iterations of the following process: find a dozen or more random goals in the literature and then modify existing classification. Each time, more goals fit into the existing classification which required fewer changes.
- The representation of low-level goals is inextricably linked with the representation of ecological and modelling concepts.
- Although all the examples have been from our test domain, the goal classification is not dependent on the ecological domain. Every occurrence of the word 'ecological' could be removed with no essential change in meaning (see figure 3–1).

Goal Usage

- The number of ways that goals may be used is fairly small.

- There is a dialogue control strategy implicit in the goal ontology graph which may be used to drive the model elicitation process in the *early stages*. With respect to this, we note that:
  - Use of goals is primarily to coax the user into saying something that is important about the model.
  - It is not the goals themselves that are useful, it is the fact that they mention specific ecological and modelling concepts which serve as hooks to guide further elicitation.
  - One role of high-level goals is to lead the user to low-level goals.
  - All dialogue paths ultimately lead to the consideration of specific concepts about the ecological system or model (*e.g.* via low-level goals)
  - Thus, acquisition of ecological modelling goals is inextricably linked with the process of acquiring ecological and modelling information.
- There is insufficient knowledge available at this time to capitalise on the other uses for goals for the *middle and latter stages* of model acquisition.

These preliminary conclusions had (and have) important implications for the direction of this research. A key observation is that in order to represent and/or acquire goals, we must first be able to represent and/or acquire specific ecological and modelling concepts. However, we can do the latter independently from goals. We therefore conclude that *there is a fundamental basis for providing assistance in the early stages of the ecological modelling process that does not necessarily depend on goals.* The reasoning goes like this: the main use of high-level goals is to help elicit low-level goals. The main use of low level goals it to extract specific ecological and modelling concepts that are used to direct the elicitation. These concepts need not be expressed in goals. For example, initially, the role of the goal "To plot temperature versus jellyfish population size?" is to identify temperature and jellyfish population size as being important concepts. The same initial effect can be achieved by having a separate mechanism to allow users to say that these things are important. It is this mechanism, not the goals themselves that is the fundamental basis for guiding model elicitation in the early stages.

However, this in no way undermines the overall usefulness of goals. We give three reasons. First, although it is not necessary to use goals initially, ecologists may find it more convenient to do so. An ecologist should be able to choose

between using this mechanism directly (*e.g.* by saying something like "temperature is important"), or indirectly via goals. In either case, what the expert does with the information is the same at the outset.

Secondly, goals are likely to play an important role in the middle and latter stages of model acquisition. Thus, ecologists should be encouraged to express goals rather than directly stating what is important. To see why, consider the goal 'What is the effect of temperature on jellyfish?'. If the ecologist states this directly, then the expert may infer that the concepts temperature and jellyfish are important. There is no need for this to be stated explicitly. If they choose not to express the goal, then they miss out on the assistance the expert might have offered (*e.g.* to ask whether the temperature affects jellyfish directly, or by some intermediate factors). If they choose to express the goal after they stated their interest in the concepts mentioned by the goal, such assistance will be offered, however they will have wasted their time when stating what was important. Had they stated the goal at the outset, the expert would already know that these things were important. This time wastage is perhaps of minor import, especially in the context of an interaction between two humans. If we simulate the expert in a computer assistant, it means unnecessary work for the ecologist.

Thirdly, from a methodological point of view the goals played a major role in identifying the important ecological modelling concepts, thus defining the requirements for expressive power.

State of Elk with respect to Goals

Although ELK currently makes minimal use of goals, they have played an important role. The most immediately useful result is the identification of the expressive power requirements for ecological and modelling concepts. In solving the problem of representing ecological and modelling concepts, we get essentially for free a representation for the most important of the low-level goals. This is a direct consequence of the fact that goals are expressed using ecological and modelling concepts. Initially, there is no need formally to represent all the goals in our ontology. We chose to limit ourselves to a fundamental subset of goals which any future version would require. We have implemented a goal elicitation assistant enabling users to specify goals of the form "What is the affect of X on Y?" and "To plot X versus Y". X and Y may be in principle any ecological or modelling

concept (*e.g.* processes, entities, attributes model variables). Independently from this, there is a mechanism enabling users to state that they are interested in certain thing, (*i.e.* that they are important). In chapter 6 we describe how such *interest specifications* could be inferred automatically from goals. This is not implemented yet.

In the medium and long term, we can implement the goal–directed control strategy as one option in a larger system. Thus, users can decide to start with high-level goals, or low-level goals, or by directly saying what the important things in the ecological system and/or model are. Having more than one way to achieve something turns out to be a key feature of ELK. The idea is that rather than forcing users to do everything in a prespecified manner, we provide options and let them decide for themselves.

The next chapter is devoted to a detailed discussion motivating and describing the design of ELK.

Chapter 4

Design Considerations

4.1 Introduction

This is the final chapter concerned with motivating and describing the design for ELK. In chapter 1 we noted four fundamental sources of difficulty for formalisation problems: syntactic adequacy, expressive power, conceptual distance, and choice management. We said that constructing a computer assistant to help overcome these difficulties consists of reformulating the original formalisation problem into a new one. In the new formulation, the same difficulties may arise in principle, but in a new form which should be easier to deal with. In the process of designing and implementing ELK, we reformulated the formalisation problem that ecologists normally face if they wish to create an ecological simulation model. This chapter describes a design rationale which argues that the reformulated version embodied in ELK will be easier to solve. We express the design in terms of requirements and techniques. Each difficulty gives rise to a requirement to overcome it. We identify techniques that may be used to meet these requirements.

Except for the syntax issue, we have already begun to do this in the previous two chapters. In chapter 2, we explored what the representation requirements are for describing ecological systems and models. A key observation was the need for a distinction between ecological information and simulation modelling information. This addressed the expressive power issue as well as clarifying the nature of the conceptual distance issue in the ecological modelling domain. The distinction between ecological and modelling information implies that we must bridge these two levels to reduce conceptual distance. In chapter 3, we discussed types and uses

for goals. They play an important role in this bridging. They also play a role in controlling choices. Thus, we have also begun to address the issues of conceptual distance and choice management. There is much more to discuss regarding these latter two due to the need to assist in the process of constructing models, and to identify and record specific modelling decisions. We will comment only briefly on syntactic issues in this chapter deferring proper treatment to chapter 7.

In presenting the material relating to the design and implementation of ELK, we face a pedagogical dilemma. The source of the problem is that although all the central ideas have been implemented, there are a number of important features that have been included in the design but have been implemented only partially or not at all. The dilemma is whether to:

- Discuss only what has been implemented which may result in a less coherent story emerging.
- Include discussion of what has not been implemented which may result in confusing the reader about what has actually been achieved.

We have chosen for the latter option in the interest of completeness and coherence. Also, because these features are included in the design, ELK may be extended in fairly straightforward ways to incorporate them. We avoid confusion by introducing bits that are not fully implemented as such and by distinguishing future plans from current implementation by using words like 'will', 'might' and 'could' rather than 'is', 'does', 'did'. More open ended extensions are discussed in § 9.3.

4.2 A Scenario

We first give a brief model-construction scenario of the sort that ELK is designed to make possible. Important services that ELK can or will provide include:

- acts as an expert modelling consultant.
- acts as an acquisition, storage and retrieval system for ecological information.
- acquires simulation model specifications.
- automatically documents models in ecological terms.
- automatically writes, compiles and runs programs.

A session proceeds in any number of ways depending on the needs of the users. If the user does not know where to begin, they will be able to click on the suggestion box which causes the system to offer a menu of very general suggestions [in some predetermined order] which include:

- specifying goals
  What is the affect of rainfall on wildebeest?
- specifying interest in general ecological concepts
  I am interested in wildebeest.
- identifying influence relationships
  Rainfall influences grass growth
- describe part of the ecological system
  wb_pop is a population of wildebeest.

Alternatively, if the user knows what to do, they will be able to do any of these things without asking for advice, but by directly selecting the appropriate command option. In any case what happens initially is that certain things about the ecological system will be either inferred or specified directly. Gradually, more and more gets specified about the ecological system. Eventually it comes time to begin specifying the model. Again, the user may ask the system for suggestions, or they may get on with it independently. The things users can do by way of defining the model include:

1. define an attribute variable in terms of an ecological attribute
   e.g. *n_wb* might be the name of the model variable corresponding to the attribute *number* of *wb_pop*.
2. define an effect variable in terms of some ecological effect
   e.g. *wb_eaten* is the annual number of wildebeest eaten by the aggregate predator population.
3. initialise state variables and parameters
4. specifying variable dependencies possibly in terms of ecological influences.
5. select or specify equations for computing variables.

4.3 Requirements and Techniques

The majority of the above scenario is possible using the current version of ELK. We discuss our design in terms of *requirements* and *techniques*. Depending on the viewpoint, other terms are useful for talking about requirements. For example, a key requirement in ELK is to have sufficient expressive power. Expressive power is an important *issue*; to not have enough is one of the fundamental *difficulties* in the formalisation process. Expressive power might also be viewed as a *feature*. We use the generic term *benefit* to encapsulate all of these other terms. It refers to any aspect of our computer assistant that is deemed to be desirable. For some benefits, the word 'requirement' is a bit strong. These include those desirable features that arose as a consequence of design decisions made to meet other requirements, but are nevertheless important. These are referred to as *fringe benefits*. All of these are incorporated into the design and at least partially implemented.

The term 'technique' refers to any construct, mechanism, or method used to help facilitate either meeting some requirement, or a more general technique which in turn facilitates meeting some requirement. In this chapter, we concentrate mostly on the requirements and mention only the very general techniques. All of these and the more specific techniques that we employ in ELK have been included for specific reasons. As we introduce each technique we justify it by making explicit how it helps facilitate other more general techniques or meeting requirements.

Everything derives from the two fundamental issues: model comprehension and model construction. From these two 'benefits', we proceed in one or more of four ways gradually getting more and more specific:

1. we give its *defining* characteristics (*i.e.* say what we mean by it),
   e.g. an adequate formalism is one that has sufficient expressive power, is easily modifiable etc.
2. we identify different examples, manifestations or *kinds* of these general benefits
   e.g. uniform representation and uniform user interface are two kinds of uniformity
3. we identify more specific benefits which serve to *facilitate* the more general

benefit.

e.g. transparency facilitates modifiability

4. we identify techniques which serve to *facilitate* the benefit.

   e.g. making explicit connections between similar concepts is a technique which helps achieve greater expressive power.

This defines a graph which characterises the design rationale for ELK. There are two kinds of nodes: requirements and techniques. There are three kinds of arcs (definitional, a kind of, facilitate). We have found that for expository purposes, the distinction between requirements and techniques is very useful, but at times somewhat arbitrary. Thus, we do not distinguish arcs as different if they happen to go between different kinds of nodes.

Figures 4–1 and 4–2 summarise the requirements portion of this graph (analogous to figures 3–1 and 3–2). Figure 4–3 relates the main techniques to the main requirements. For readability, a number of useful distinctions are blurred in figure 4–2. For example, there are no 'a kind of' arcs (*e.g.* in § 4.5 we note several kinds of consistency checking). The 'a kind of' relationship also arises where a benefit applies both to representation issues, and assistance/interface issues. These include:

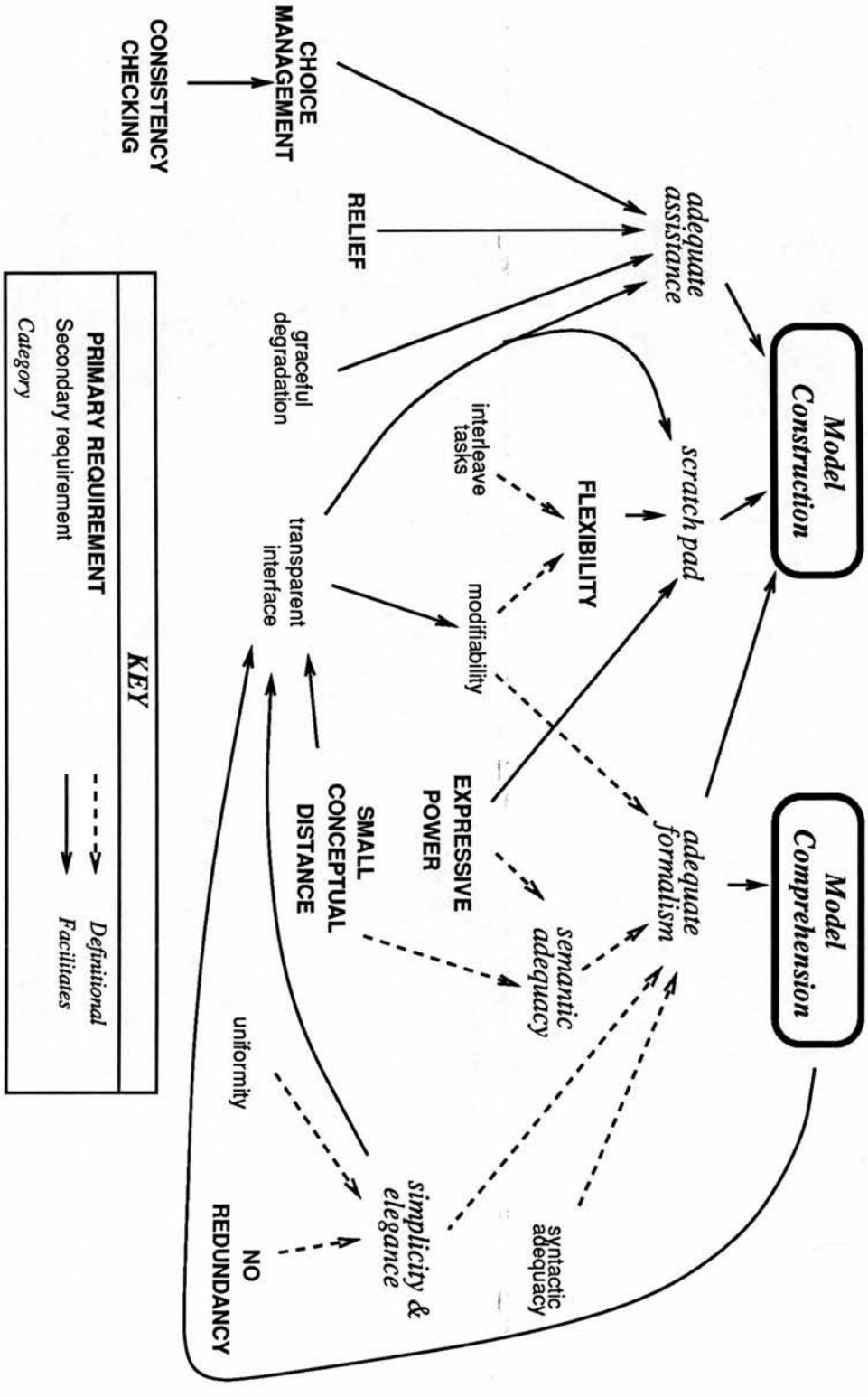
1. conceptual distance
2. comprehension/transparency
3. modifiability
4. simplicity and elegance
5. uniformity
6. no redundancy
7. few primitives

For example, when we speak *conceptual distance with respect to the interface*, we mean that the primitives that the interface provides map closely to how users think about their problems. When we speak of *conceptual distance with respect to the representation*, we mean the semantic primitives provided by the underlying formalism match the users terms. An important relationship constraining the design of ELK is that reducing conceptual distance in the representation can be a great help in reducing conceptual distance in the interface. Also, although related, lack of redundancy in the representation means something rather different than lack of redundancy in the interface.

1. *adequate representation formalisms*; by which we mean:
   (a) *syntactic adequacy*
   (b) *semantic adequacy*; by which we mean:
      i. *expressive power*
      ii. *small conceptual distance (representation)*
   (c) *simplicity/elegance (representation)*; by which we mean:
      i. *no redundancy (representation)*
      ii. *uniformity (representation)*
   (d) *modifiability (representation)*
2. *adequate assistance*; This is facilitated by:
   (a) *choice management*; this is facilitated in a major way by:
      i. *consistency checking*
   (b) *transparent interface* (*i.e.* easy to understand)
   This is in turn facilitated by:
      i. *small conceptual distance (interface)*
      ii. *model comprehension*
      iii. *uniformity (interface)*
   (c) *relief from redundant tasks*
   (d) *graceful degradation*

3. use as a *scratch pad*; This is facilitated by:
   (a) a *flexible interface*; by which we mean:
      i. easy to *modifiability (interface)* (*i.e.* ability to change your mind)
      e.g. redo, undo
      ii. easy to *interleave tasks* as desired (*i.e.* user initiative)
   (b) *transparent interface* (≡ item 2b)
   (c) *expressive power* (≡ item 1(b)i)

Figure 4–1: Elk Requirements



This shows the major requirements driving the design of ELK *and the important relationships between them.*

Figure 4–2: Design Rationale: Requirements

Figure 4–3: Elk Design Rationale

It is worth distinguishing between the different manifestations of these issues with respect to the interface versus representation issues. This is because the requirements that they facilitate, and the techniques that facilitate them differ, as well as relationships between them holding as for conceptual distance. They are worth referring to collectively because general relationships hold between them; *e.g.* transparency facilitates modifiability.

Most of the above 7 items are self explanatory, however there are some important relationships that hold between them. For example, *comprehension*[1] is facilitated by *simplicity and elegance*; *uniformity* and *lack of redundancy* are defining characteristics of simplicity and elegance. It is facilitated by having relatively *few primitives*, each reusable.

All of the requirements facilitating one or both of model construction and model comprehension come under three categories: adequate representation formalisms, adequate assistance, and ability to use as a scratch pad. The first two are fundamental, and are addressed separately in sections 4.4 and 4.5 respectively. We discuss the third here.

Given the open ended nature of the modelling exercise, it was deemed that the system should be able to be used as a *scratch pad.* To facilitate this, we require a *flexible* and *transparent interface*, as well as ample expressive power. Relating to the latter, there is little to be gained by providing a scratch pad if there is hardly anything to scratch with! Expressive power is a major requirement which is dealt with in § 4.4.

By flexibility, we mean two things: *modifiability in the interface* (*i.e.* ability to change your mind; redo, undo, etc.) and the ability of the user to *interleave tasks*, (*i.e.* doing things in any order). The major facilitators of modifiability in the interface are *transparency in the interface*, and *modifiability in the representation.* The chief technique used to meet the latter requirement is implicit specification (see § 4.4.4). This makes it much easier to incorporate *undo* and *redo* features, many of which are in the current implementation, at least in rudimentary form. The transparent interface benefit is discussed in § 4.5.

---

[1] When a benefit or technique is first or most prominently introduced or characterised in the text, we put it in *italics.*

Before we get on with the major categories, we comment on efficiency. The nature of this formalisation exercise is such that computational speed has not been a major concern. Because the user will spend a fair amount of time thinking, for any single step in the formalisation process, there is relatively little computation that is required. This is an important point because it has allowed fairly free and uninhibited addition of expressive power to the formalism.

Space efficiency is a fringe benefit. It derives from a general *lack of redundancy in the representation*, which is made possible by the technique of implicit specification. By this we mean inference is used rather than having to specify everything explicitly. This technique is embodied in various inheritance and inducing mechanisms.. Other examples of implicit specification are given in § 4.4.4.

4.4 Adequate Representation Formalisms

There are various important issues to consider regarding formalisms:

1. syntactic adequacy
2. expressive power
3. conceptual distance of the representation
4. lack of redundancy in representation
5. uniformity in representation
6. modifiability of the representation

The first three correspond to the fundamental difficulties of the formalisation problem outlined in chapter 1; these will be considered separately. The latter three are discussed together (§ 4.4.4); they are desirable features of any representation. Note that there is a distinction to be made regarding whether the benefit is to the end user, or to the system developers, or both. Items 1 and 2 are direct benefits to the end users; the system developers have to work to achieve them. Items 3 and 6 are best viewed as making life easier on the system developers. Small conceptual distance and modifiability in the representation respectively facilitate keeping the conceptual distance with respect to the interface primitives small and provision of features for allowing users to change their mind. Items 4 and 5 are more directly useful to system developers, but indirectly useful to end users.

4.4.1 Syntactic Adequacy

In the reformulated version of the ecological modelling formalisation problem embodied in ELK, the issue of *syntactic adequacy* does not refer to how our underlying formalism looks. Rather, it refers to the set of user interface commands available to the user. The syntax of our formalism is not meant for human consumption in its raw form. Thus we have provided a sugar coating using interface mechanisms that enable each construct in our formalism to be created or examined with minimal effort. It is this that achieves syntactic adequacy in the reformulated formalisation problem. We can do this with minimal conceptual effort because we have a fairly *uniform representation* and small conceptual distance with respect to the formalism. Facilitating the eventual incorporation of very slick interface features was a high priority in this project. However, our current implementation is fairly rudimentary in this regard, serving only the purpose of demonstrating the basic concepts.

4.4.2 Expressive Power

By *expressive power*, we mean the ability to represent and reason about the required information and distinctions in a domain. We regard the notions of expressive power and 'richness and generality' as roughly equivalent. It is important to note that that more expressive is *not* necessarily better. There is nothing gained by making distinctions that are not likely to be used. Rather, our basic philosophy is:

> *to 'properly' represent concepts that are likely to be used frequently by equipping the system with useful knowledge about that concept*

The usefulness of a richer, more general representation is manifest in a myriad of ways. Some of the more important of these include consistency checking, search space control, and provision of explanation and automatic documentation facilities. The idea is, the more the system 'knows', the more services it can provide.

A basic question faced both by the formalism designers and ecologist users is:

> *For a particular concept, how rich and general does the representation for it need to be?*

Consider how we might represent the concept of a black sheep. There are several concepts implicit in this including sheep, colour, animals, and physical objects.

The question is when and whether all this is useful to incorporate explicitly in the representation. We list five possible representations in order of increasing richness and generality. In each case, we note what concepts the system does and does not 'know' about (recall that $E{:}T$ means the entity E is of type T).

1. *black_shp* : *animal*
   System knows only about animals, not colours, or sheep.
2. *black_shp* : *sheep*
   System knows only about sheep, not colours, and not black sheep
3. *shp*(*black*) : *sheep*
   System knows about sheep, colours, and that black is a colour, but not about black sheep as a separate type.
4. *black_shp* : *black_sheep*
   System knows about black sheep as a special type, but not about colour; may or may not know about sheep, if so *black_sheep* $\sqsubset$ *sheep* should be true.
5. *black_shp* : *sheep*(*black*)
   System knows about sheep, colours, and that black is a colour, and black sheep is a subtype of sheep.

Balance must be struck between cost of extra complexity versus frequency of use. The extra complexity has two edges:

- extra machinery in language and corresponding processing procedures (relevant to system developers)
- extra work in making and understanding specifications (relevant to users)

There is no point in having types that are used infrequently. It may be adequate to incorporate certain information only in the names. In the first case, the system knows nothing about 'sheep', 'colour', or 'black', only animals. It is always the onus of the user to manually keep track of what the system does not know about. This may be no burden if these concepts are not frequently used. In the final case, the system knows about the type *sheep*(*black*), which is a subtype of *sheep* derived on the basis of the value *black* of the attribute *colour*. The relationship between *sheep*(*black*) and *sheep* is formally the same as that between *sheep* and say, *phys_obj*. In designing the system, we deemed that the first four cases should be catered for. However, the latter case would exact costs of both the above kinds

which would not likely be offset by the extra benefit. Empirical evidence will cast light on the adequacy of this decision. Even when the system provides all the facilities to give maximal generality, there is still an option for users as to whether the bother of using the extra machinery is justified. Whether the decisions is being made by the designers or users of the formalism, the same criteria of frequency of use is the major determining factor.

Our achievements with respect to expressive power are mostly due to the technique of *retaining connections* between similar concepts.[2] We discussed this in chapter 2 with reference to the attribute concept, 'number' as applied to various populations. In ELK, the 'number' concept is known to the system as an attribute that applies to all and only set entities. This technique serves to eliminate redundancy because the user never has to invent a multiplicity of similar names like *n_wb*, *n_prey*, etc. Similar comments apply to the above example with the concepts of colour, sheep, etc. Supporting this general technique of retaining connections are three key features of our formalism:

1. *few representation primitives* compared to the number of expressions possible.
2. ample *use of functions* for combining primitives to form complex expressions.
3. *rich type structure*

These three features together facilitate two additional things: implicit specification and model comprehension. These work in conjunction with the above three techniques. Implicit specification in turn contributes to lack of redundancy and space efficiency as noted at the end of § 4.3. Very importantly, because each primitive as well as each combining function has ecological meaning, *semantic translation* of complex expressions into ecological/modelling terminology is a straightforward exercise. This in turn, facilitates one of our major requirements: model comprehension. There are other important uses for the technique of retaining connections (and supporting techniques) that are described in § 4.5.

---

[2] It is equally valid to view 'retaining connections' as defining expressive power; how we choose to talk about this does not affect our main points.

4.4.3 Conceptual Distance

Achieving *small conceptual distance with respect to representation* primitives was a major design constraint. It cannot be separated from the process of identifying the types of knowledge that were required and subsequent design of appropriate constructs to represent them. A wide spectrum language was developed to bridge the conceptual gap between information purely for representing simulation models and purely ecological information for describing the ecological system of interest. In the next section we discuss in greater detail, the additional types of knowledge which bridge the gap. The successful design of a formalism to satisfy our constraints was the single most important (and largest) task of this research. It impacts on virtually all of the major requirements and serves to bind the two major categories of design requirements.

There are no more basic techniques which facilitated achievement of small conceptual distance with respect to the representation, however it facilitated the following key benefits.

- *model comprehension*: this is because it only requires direct translation of the expressions in the intermediate formalism rather than complex interpretation.
- *small conceptual distance of the interface primitives*: this is because the interface commands map directly onto the formalism. It makes it easier to design the interface, the details of which would be unknown to users.

4.4.4 General Requirements

Finally, we consider three very important general design requirements which apply to any knowledge representation exercise. Chief among these is simplicity and elegance. By a *simple/elegant representation* we mean one which is *uniform [representation]* and has *no redundancy [representation]* (or more accurately does not give rise to redundancy when used). The general technique of retaining connections and its supporting techniques discussed above are the major source of simplicity and elegance in our representation.

A simple/elegant representation facilitates semantic translation which in turn facilitates *representation comprehension.* By this we mean it is easy to determine what an expression means. It directly facilitates mechanisms for *automatic docu-*

mentation which help achieve model comprehension as noted above. Representation comprehension also facilitates *modifiability [representation]* and *extendability.*

We note one additional technique which facilitates modifiability: *implicit specification.* The simplest example of this is the inheritance of attributes. Another example is the representation of the transitive closure of a relation. For example, if *twig*1 is a component of *branch*1, and *branch*1 is a component of *tree*1, then by transitive inference, we implicitly specify that *twig1* is a component of *tree1.* A third example is the inducing of attributes as discussed in § 2.5.5.1. Our philosophy is to keep as many specifications as possible implicit. This is very important because it means that there is zero work in updating them when changes are made. They update themselves automatically. It also saves space. There is a tradeoff with computational speed which at times must be dealt with, but this has so far not been a significant problem.

4.5 Adequate Assistance

The major issues which concern us here are listed below (roughly in order of decreasing importance):

1. model comprehension
2. conceptual distance (interface)
3. choice management; this is facilitated in a major way by:
   (a) consistency checking
4. relief from redundant tasks
5. graceful degradation
6. uniformity (interface)

Besides being one of the major objectives (in this thesis) in its own right, meeting the model comprehension requirement also lends strong support to the other major objective: to alleviate difficulties with model construction (see figure 4–2). Issues 2 and 3 correspond to the fundamental difficulties of the formalisation problem. The others are desirable features of any interface, and although important for us, not as fundamental as the first three. Here the benefits are exclusively to the users, not the system developers. In the following sections, we elaborate on each of the above benefits.

4.5.1 Model Comprehension

Model comprehension corresponds to the ability of users to understand a simulation model in terms of the system being modelled. There are two main aspects. First, each component of the simulation model (*e.g.* variable, equation, parameter) is accountable for in domain terms. Second, it requires explicit identification of the assumptions and simplifications that are embodied in the model.

In the context of ecological modelling, given the modelling primitives described in chapter 2 achieving model comprehension means being able to represent [at least] the following information:

1. The ecological meaning of every variable is given.
   - *e.g.* *n_wb* denotes the number of wildebeest.
   - *e.g.* *wb_eaten* denotes the total annual number of wildebeest eaten by predators eaten per year
2. Simplified value spaces
   - *e.g.* biomass has values *large* and *small* rather than reals.
3. Aggregations
   - predators consists of lions and hyenas
4. What is potentially important but is ignored for the simulation model.
   - migration is important, but ignored
   - predation causes biomass of wildebeest population to decrease, this is not of interest.

To complete the job, mechanisms for automatically producing English text documentation of the model are required. The key features in our formalism which render this task an easy one are the rich type structure, relatively few primitives, and ample facilities for combining these primitives into complex ecologically meaningful expressions. Examples are given in chapter 6. In ELK there is a considerable amount of such text generation but there is scope for much more. This has received low priority for the same reasons as making the interface super-slick did. It is a time consuming job with few conceptual difficulties.

Achieving model comprehension in turn facilitates transparency in the interface in that, the user will be able to easily see the results of what they have constructed. This in turn facilitates both adequate assistance, and use of ELK as a scratch pad; each plays a major role in addressing the model construction issue (see figure 4–2).

Importantly, *the exact same techniques that facilitate model comprehension, also facilitate reduction of conceptual distance, and help manage choices.* The key technique used to help achieve model comprehension is to represent both domain information and the simulation model separately and to have explicit links between the two. Development of a language to represent this information is exactly what is required to reduce conceptual distance. It also identifies the modelling search space, a great help in managing choices.

4.5.2 Conceptual Distance

Along with model comprehension and search control, reducing conceptual distance to enable computer naive ecologists to construct models by communicating in their own language has been a parallel central focus of this research. We had to bridge the gap between ecological information and a simulation model. The best way to ensure this is to keep the conceptual distance in the formalism primitives small. Here we explore what those primitives (*i.e.* language constructs) need to capture.

Our approach is based around a single key technique: *gradual elaboration*, (*i.e.* incremental specification). By this we mean users should be able to start with specifying small simple concepts and gradually elaborate on them. From the point of view of the final runnable model, these may be highly vague. From the point of view of the system, these are specific bits of information that are used to focus the entire process of model elicitation.

Gradual elaboration is manifest in two ways corresponding to two sort of gaps that need to be bridged.

1. from ecological concepts to modelling concepts (*i.e.* moving from one level of information to another)
2. from simple to complex items in a specification. (*i.e.* within the same level)

To facilitate gradual elaboration and successfully bridge the conceptual gap, we require first and foremost the ability to describe ecological systems and models separately (as discussed in chapter 2). However we require additional bridging information as well. We have seen that *goals* play an important bridging role. They serve as hooks to guide and focus the formalisation process in the early stages. Low level goals refer explicitly to ecological and/or modelling concepts which by their very mention suggests that certain things are important, and thus that the

user is *interested* in them. Goals also imply influences and suggest certain computational dependencies. For example, the goal: 'What is the effect of rainfall on the wildebeest population' suggests that rainfall influences the wildebeest population. Because influences are idealised as computational dependency, this information can be later used to constrain choices during equation formation when choosing schemata to compute model variables (see [Robertson et al, 1987]). In this case, the equation for computing [some of the] variables representing the wildebeest population should depend on the variable representing rainfall. The system could ask if there are any intermediate influence relationships.

So, in, general goals can be used by the system to suggest any number of further courses of action to take; *e.g.* to elaborate on the ecological concept, or to specify how it is to be represented in the simulation model. This is an important technique in reducing conceptual distance.

Goals and interest information give rise to two classes of constructs in our formalism. They are both examples of specifications whose meaning and/or role is neither to describe part of the ecological system, nor to describe part of the model. Rather, they serve the purpose of informing the interpreter (read 'intelligent dialogue manager') how to guide the formalisation process itself. This kind of construct in conjunction with a range of constructs for directly specifying ecological concepts (*e.g.* entities, attributes, influences) are the key techniques by which we reduce conceptual distance and facilitate gradual elaboration.

Note that putting anything at all in the description of the ecological system automatically implies that the user is interested in the thing. We have not yet incorporated the automatic creation of interest specifications, but the various hooks are all in place to facilitate this. The biggest question is to decide what the best sort of support the system should offer is without becoming a nuisance. This is but one example of a general facility that we provide for automatic creation of pieces of the specification. This is discussed in § 4.5.4.

After the ecological system has been described, users must specify what aspects need to be modelled and how. A user should be able to specify:

- that a specific attribute of a specific ecological entity
  - is a state variable in the model.
  - is modelled with a specific value space.

- that an ecological state variable is inc/decremented by a partial rate variable corresponding to the effect of some process.
- what the initial value for the state variable is.
- that one variable depends on another.
- that a variable is computed using a specific equation.

These things are all readily understood modelling concepts. It is the job of the system to convert these specifications into the appropriate logic expressions which define the simulation model. Constructs which facilitate the input of such information are described in chapters 5 and 6.[3]. They contribute significantly to the achievement of the goal of keeping the conceptual distance for the interface primitives small. An additional important feature which makes the interface able to communicate effectively in ecological terminology is the semantic translation machinery which generates English text from specification constructs.

4.5.3 Choice Management

In chapter 1 we distinguished three levels of assistance (*identify*, *prune*, *advise*) that may be provided for making two kinds of choices (what to do and how to do it). From the user's point of view, identification of the choices of what to do is implicit in the set of available interface commands. Almost all of the interface commands in ELK are one of two basic types: 1) edit the specification and 2) look at the state of the specification. Because of the flexibility requirement, it is rare that we wish to prune choices by forbidding certain commands. We would, however, routinely prevent certain ways of using a command that do not make sense (especially in updating the specification). However, through the interest specification mechanism, we provide some advice in the form of giving suggestions about what might be a good idea to do next. It is not possible to do a lot more than this without an extensive knowledge acquisition exercise perhaps in conjunction with empirical work to determine what control strategies are better than others. It is our philosophy to be non-intrusive and let users do things in any order that they see fit, so long as their specifications are consistent.

---

[3] This is true except for explicit variable dependency. See § 9.3 for further discussion of the state of the implementation.

We provide rather more assistance on the issue of how to do things. Many 'how' choices arise in the process of constructing ecological simulation models. Roughly corresponding to the two stages discussed in § 4.5.2, we identify two search spaces which must be navigated by users. First, there is a vast space of possible ecological systems that could be described. Second, there are many ways that a particular ecological system may be idealised in a simulation model.

A major part of this research has consisted in *identifying what these search spaces* were, (particularly the latter) and how they may be formally represented and manipulated in a computer assistant. In solving this problem for a significant portion of the ecological and modelling domains, we are in a position to provide the most basic form of assistance in managing choices: identification. For example, the list of modelling specification options given at the end of § 4.5.2 is part of the identification of the simulation modelling search space. From the point of view of assisting an ecologist with limited modelling skills, this constitutes a very important first step. However, we do more than this. We also use *consistency checking* to heavily prune the search space. This serves three purposes simultaneously. First, the blank sheet of paper syndrome goes away; next, fewer choices are more manageable, minimising need for browsers; finally users are prevented from describing nonsense which is important in its own right. The pruning is of course invisible to the user.

On the whole, we provide very little advice on what the best way to do something might be. To do so requires a significant challenging knowledge acquisition exercise which gets at the heart of how people build models.

Consistency Checking

The need to maintain consistency in the ecological system description gives rise to another important distinction in the information that our system requires. If we wish to ensure that descriptions of ecological systems are consistent, we must appeal to a separate level of ecological knowledge. This suggests the following division:

- the *general/ecological* level consists of knowledge that is accepted to be universally true.
- the *ecological system* level consists of a description of the specific ecological system being modelled (real or hypothetical).

To describe the consistency maintenance machinery, we introduce a notion of *permission*, an important requirement of our formalism. In order to specify something as part of an ecological system, the general/ecological knowledge base must permit it. The idea is to prevent saying that some population of lions eats blue whales, or that branches are parts of elephants. This notion is also used to help prevent inconsistent models from being created. For example you cannot create an ecological model variable which corresponds to the biomass of a rock, or the number of members of an elephant. Nor can you say that the process of predation is going to be modelled by increasing the state variable representing the biomass of the prey. The majority of this consistency checking is facilitated by the rich type structure in the formalism in conjunction with many rules. Thus, much apparently semantic consistency maintenance of this form is achieved using syntactic methods. Summarising, there are two stages of consistency checking:

1. *Ecological systems must be consistent* with general/ecological knowledge
2. *Ecological simulation models must be consistent* with the ecological system.

Distinguishing between the general/ecological and ecological system levels is required to achieve the first kind, and distinguishing between the collective ecological levels and the modelling level is required for the latter kind. The latter distinction is also fundamental for facilitating model comprehension.

Another useful sort of consistency checking is *completeness checking.* Given a notion of what it takes for specification to be complete, identification of what is missing can serve to provide assistance by identifying what the user needs to do next (possibly via suggestion box, or agenda). In [Robertson et al, 1988b] this form of assistance was referred to as gap-filling. Although no such mechanisms have yet been incorporated into ELK, it would be relatively easy to do so.

A crude but useful notion of completeness of the simulation model is when every output variable can be computed. Computability of a variable is a recursively defined notion which goes roughly as follows. A parameter is computable if it is initialised to some value. An exogenous variable is computable if a procedure has been given which returns a value for any given simulation time. A variable is computable if every variable which is a direct input to the function that is used to compute it is computable. This technique is used in SL [Robertson et al, 1988a] and NIPPIE [Haggith, 1990] to guarantee runnable simulation models. These programs are part of the ECO project and are described in § 8.2.1.3. Variables must

always 'bottom out' to parameters or exogenous variables. The latter are by definition independent of any model variable.

4.5.4 General Requirements

We now consider three general requirements which are useful in any interface. These are: *relief from redundant tasks*, *graceful degradation*, and *uniformity [interface]*.

Relief

We have provided a wide range of facilities to make life easier for users by preventing the need for many menial and/or repetitive tasks. This is accomplished by offering considerable scope for *reuse* of information in one form or another. There are several major aspects to this:

- implicit specification
- defaults
- reuse (high-level)

Implicit specification keeps users from having to specify over and over minor variations of the same concepts (*i.e.* relief from menial/redundant tasks). *Taxonomies* and induced attributes play a key role here. This is the basic principle of reusable primitives described in the section on expressive power.

Another facility for saving users a lot of hassle is in the provision of *defaults*. We distinguish two kinds: those specified by the user, and those specified by the system. To make this clear we use an example. In designing or augmenting the general/ecological knowledge base the attribute 'weight' might be specified for physical objects, a type of ecological entity. This would include specifying a value space, say real numbers. Now, every time a model variable is created which corresponds to the weight attribute of some physical object, by default the value space for the model variable will be inherited from the specification in the general/ecological knowledge base. This is a *system specified default*. This may be overridden for an specific instance, say using the value space $\{small, large\}$. However, if it should turn out that there were several weight model variables and the simpler value space was wanted for them all this would be a nuisance. It is not an acceptable solution to change the value space to $\{small, large\}$ in the general/ecological knowledge

base for two reasons. First, the decision about using large and small is a modelling decision, so it does not belong in the general/ecological knowledge base. The second reason is purely practical. There might still be other physical objects that you wanted to have the usual value space for.

Instead, we provide a class of specification constructs for *user-specified defaults.* The user could use a construct of this kind to specify that "every time a model variable is created which corresponds to the weight attribute of a physical object, then the value space will by default be $\{small, large\}$". We have designed and partially implemented a facility for selectively causing automatic creation of specification constructs in a pre-specified way. For instance, the user can specify that "every time an entity of type *sheep* is put in the ecological description, then automatically create a model variable corresponding to the weight of that sheep entity." There are other things that may be specified as defaults; these are described fully in chapter 6. These separate constructs used in this way enable us to achieve the desired relief from redundancy without sacrificing the integrity of the general/ecological knowledge base.

These constructs, like goals and interest specifications are neither part of the description of the ecological system, nor are they part of the simulation model description. These are merely directives to the dialogue manager to fill in defaults in a certain way. We say that such information is at the *dialogue level* which is a sub-level of the simulation modelling level.

Finally, we discuss reuse at a higher level of abstraction. The distinction between the two ecological levels and the simulation modelling level means that one general/ecological knowledge base may be used many times to guide and constrain the description of many different ecological systems. Also, the same model constructs and schemata may be used to describe different simulation models. Finally, many different simulation models of the same ecological system may be created and compared (by the same or different users).

We have designed ELK to be used in *various distinct modes* by the same or different users. One mode is the construction of general/ecological knowledge bases. Another is to define, save, and retrieve any number of real or hypothetical ecological systems based on some general/ecological knowledge base. Finally, for each ecological system described, one or more different simulation models may be specified. The latter is particularly important; it facilitates *experimentation with*

model structure. This can be used to answer such questions as "What is the affect of ignoring this component of the model?". This is one kind of modelling goal.

Catering for the ability of users being able to extend and modify the general/ecological knowledge base was an important design constraint. It is recognised that no matter how many things were in the knowledge base, we could not hope to include everything. To ensure that this facility worked properly, the system was used to bootstrap itself (successfully).

Graceful Degradation

By graceful degradation, we mean that when the user makes mistakes, they should not be left hanging. Rather, the system should offer suggestions about remedying things; possibly taking over initiative from the user for a short time. Ideally, a suite of *recovery mechanisms* should be provided which use the same facilities required by the [as yet not implemented] agenda mechanisms. This is provided to a limited extent, but there is much scope for extending this. The recovery mechanism idea is to some extent just a frill, although a very useful one when taken as a whole package. Again this is conceptually unrewarding. That it is straightforward is largely due to the typing. Because the type checking catches the error, the nature of the error is already known making it easy to suggest how to put it right. Specific examples are given in chapter 7.

Uniformity in the Interface

Uniformity in the interface primitives is little more than common sense in interface design. We have attempted to use similar techniques for accomplishing similar tasks wherever possible.

4.6 Knowledge Levels

The most important aspect of our design is the ontology of information/knowledge. There are three main reasons for distinguishing between the ecological level and the simulation modelling level (the latter is less a central design constraint than the others):

1. to ensure model comprehension

2. to identify the idealisation search space
3. to facilitate experimentation with different models of the same ecological system

We further distinguish two kinds of ecological information. General/ecological knowledge is accepted to be universally true about the world in general or about ecology in particular. For example, that there are physical objects is general knowledge, that there is a process of predation is ecological knowledge. The second category consists of a description of the specific ecological system being modelled (real or hypothetical). For instance a modeller might be concerned with actual sheep in his field, or about the specific wolf population that is preying on them. This further distinction has three main benefits:

1. to ensure that ecological descriptions make sense
2. to identify and prune the search space for describing ecological systems
3. to facilitate reuse

The primary benefit is to ensure consistency. We wish to prevent users from saying that worms prey on elephants. We also wish to stop them from specifying that there are wazoolas in the ecological system unless a wazoola is a meaningful entity that the system knows about. If the system should know about wazoolas, then users simply add them to the general/ecological knowledge base (possibly prompted by the system). This entails adding it to the type hierarchy, specifying what attributes it has, and saying what if any parts it has or composites it is part of. If the system does not really need to know about wazoolas, but the user fancies having one anyway, they may create an instance of the most specific entity in the hierarchy which a wazoola is a kind of and create an instance of that entity (say animal) and give it the name 'wazoola'.

The latter two benefits go hand in hand. Since the general/ecological knowledge is intended to contain only universally true information, it can remain fixed and be used to define a large number of different ecological systems by the same or different users. This is reusability. Each time a new ecological system needs to be described, the fixed general/ecological knowledge base effectively defines the search search space for describing ecological systems. Ensuring consistency prunes this search space.

We further distinguish two kinds of simulation modelling information. The *runnable-model level* consists of information that is required to run the simulation. This includes model variables, initial values, and equations. The *dialogue level* consists of information whose role is to give hints or explicit directives to the dialogue manager which are used to guide the modelling process. We have identified various type of information of this sort. These include goals, interest, user-specified defaults, and variable dependency information. None of this information can be viewed as saying something that is generally true, nor that something is part of the ecological system; neither is it required as part of the specification for the runnable model. It may not be immediately obvious that variable dependency is not required for the runnable model. We mean that statements of the sort "X depends on Y"are not required *explicitly.* Of course if in the runnable model, Y is used in the equation that computes the value for X then this dependency may be inferred from the specification of inputs and outputs. Explicit statements of variable dependency could be used to constrain the possible equations to select for computing some quantity.

The benefits of having both kinds of simulation modelling information are many. The runnable-model information is necessary for obvious reasons. The dialogue information goes a long way towards providing useful assistance during the modelling process. This includes facilitation of gradual elaboration, reduced conceptual distance, relieving the user of menial tasks etc. Keeping these levels separate is useful primarily to enhance conceptual tidiness, clarify exposition, and simplify the implementation. From a practical standpoint it is not substantially more useful to distinguish between the two simulation modelling levels as it is between each kind of information within the dialogue level. Each kind of information is used in specific ways by the system.

For a simple model one might argue that there is no need to define everything twice which is what we effectively require by having separate ecological system description from the model. However, we escape this criticism by having an extensive facility for specifying defaults. Thus if a user desires, they can set up the system so that every ecological attribute they define automatically results in a model variable. So the user does no extra work in defining the model, but they reap benefits in two major ways:

- the model is automatically documented

- they have the flexibility to experiment with the model while the ecological system description remains constant.

The many distinctions we make increase expressive power, but also the potential for confusing users. The design aims to avoid such confusion by allowing users to exploit only those features that they require. In doing so, they are not forced to understand the underlying complexities. In the next three chapters we substantiate these claims by:

1. describing the details of the formalism
2. giving examples which demonstrate the range of ecological systems and models we can describe with it.
3. describing how the system works by going through a portion of a model elicitation session with ELK.

4.6.1 User versus System

It is important to realise that for each of the above levels the information may either be dynamically specified by the user, dynamically specified by the system, or permanently resident in the system. For example, general/ecological knowledge permanently resident in the system includes the fact that *number* is an attribute that applies to all and only set entities, as well as the notion of sets in the first place. The system will have some basic entities like animal and plant. The user will usually specify a number of entities themself. At the ecological system level the user will create instances of entities, but the system will automatically inherit attributes for them using the general/ecological knowledge base. Users will never directly specify attributes at the ecological system level.

At the runnable-model level users will create model variables; they may exploit user specified defaults and tell the system to automatically create model variables in specific circumstances. At the dialogue level, the vast majority of dynamically specifiable information will be directly specified by the user. One possible exception that we noted is the automatic creation of *interest* specifications on the basis of something being put in the ecological system, and/or as part of a goal. Additionally, there is a potentially large amount of static guidance knowledge used by the dialogue manager. Currently most of this is hard-wired (*e.g.* suggesting that an entity in the ecological system should have some attribute as model variable).

In the future, this could consist of potentially a huge knowledge base of rules and guidelines.

4.7 Summary and Conclusion

This completes the discussion of the design rationale. A clear picture of the design of ELK is emerging. The most important issues constraining the design (in approximate order of decreasing importance):

- model comprehension
- expressive power
- conceptual distance
- choice management
  - identify modelling search space
- consistency checking
- flexibility
- relief from redundant tasks

The major techniques that support one or more of these requirements are noted below.

- retain explicit connections
- few representation primitives
- use of functions for combining primitives to form complex expressions
- rich type structure
- gradual elaboration
- distinct information/knowledge levels
  - ecological
    - general/ecological
    - ecological system
  - simulation modelling
    - dialogue
    - runnable model
- automatic documentation
- implicit specification

- defaults
- user-driven dialogue

Figure 4–3 gives a graphical depiction of the key relationships between the major requirements and the major techniques to meet them.

Conclusion: Part I

This chapter concludes the first part of this thesis. We have:

- defined the general problem of formalisation
  - identified major difficulties
  - outlined solution approaches to these difficulties
  - stated a goal hypothesis
- defined a particular problem of formalisation: ecological modelling
  - characterised the domain
  - identified major difficulties
  - outlined solution approaches to these difficulties
  - tested our goal hypothesis in the domain of ecological modelling

In doing so, we have carried out a thorough requirements analysis for ELK, a computer assistant for ecological modelling. The key aspect of the design is the multi-level knowledge/information ontology. This gives rise to two additional major hypotheses of this thesis. The basic question that we wish to explore is whether we can design a formalism such that each construct can be given an unambiguous interpretation which can be used to query the system within and across each of these levels. Expressions at the runnable-model level should be related to, understood in terms of, and consistent with expressions describing the ecological system. Similarly, expressions describing the ecological system should be related to, understood in terms of, and consistent with expressions describing general/ecological knowledge. Expressions at the general/ecological level should be related to, understood in terms of, and consistent with the world as we understand it. Also, the dialogue level constructs should facilitate reduction of conceptual distance via a process of gradual elaboration, as well as help relieve the user of menial tasks. Our next two major hypotheses are:

That building a computer assistant based on our knowledge ontology can help achieve the benefits that we claim it can in the context of ecological modelling.

That every piece of information that is deemed to be useful in the process of constructing ecological models can be unambiguously placed into our ontology.

We refer to these as our *ontology usefulness* and *ontology completeness* hypotheses respectively. We show that the first is true. Our experience is that the second is nearly true, but we certainly do not expect a 100% hit rate. Rather we believe that by pushing this as hard as possible, we will identify fuzzy cases which will shed light on the ecological modelling process. This, in turn will facilitate increasing the competence of our computer assistant.

The next part of this thesis is devoted to testing these two related hypotheses. In the final part of the thesis, we discuss the relevance of this work to other research, and the extent to which our techniques may apply outside the ecological modelling domain.

Part II

A Solution

Chapter 5

ElkLogic

5.1 Introduction

In chapter 2 we described a simple model of part of the Serengeti ecosystem. We noted some problems with the representation of that model. Most of these are ramifications of the fact that the expressive power of the representation is inadequate for our purposes. It is geared only to representation of runnable simulation models and contains no explicit ecological information. Among other things this means that:

- there is no account of the model in ecological terms
- it cannot support a wide variety of consistency checking
- there is a proliferation of one-off primitives (*e.g. n_wb, n_pred*).

We identified four levels of information which constitute our knowledge ontology. Below we list the important predications at each level that our formalism must be able to represent.

General/Ecological Level: for expressing what is true in the world, (*i.e.* general truths).

- *existence of a type of entity:* there are such things as wildebeest
- *taxonomic information:* a wildebeest is a type of animal
- *existence of a certain type of substructure relationship:* A set of animals can be a subdivision of an aggregate animal population.
- *definition of an attribute:* 'number' is an attribute that applies to set entities and is integer valued.

- *definition of a process*: predation is a process that takes place between two types of animal entities, one is called the predator, the other the prey. Predation transfers biomass from the prey to the predator, decreases the numbers of the prey population etc.

Ecological System Level: for expressing what is true in the particular real or hypothetical system being modelled.

- *existence of a specific entity:* there is a wildebeest population in the ecological system that is to be modelled. It is called *wb_pop*.
- *a specific substructure relationship:* the lion population is a component of the [aggregate] predator population.
- *an occurrence of a process:* the predator population is preying on the wildebeest population.

Dialogue Level: for expressing what is true about the modelling exercise.

- *a goal:* The user is interested in discovering what the affect of increased dry-season rainfall on the wildebeest population is.
- *something in the ecological system is important:* the 'number' attribute of the wildebeest population is important.
- *default specifications:* every ecological model variable that corresponds to the attribute 'capture coefficient' of an animal population is represented as a parameter by default.

Runnable Model Level: for expressing what is true in the simulation model.

- *definition of a model variable:* n_wb is the model variable representing the simulated number of wildebeest in the population.
- *initial value of a model variable:* $n\_wb_{init} = 34000$
- *equation eq_A is used to compute the value of variable grs_wt*

In this and the following chapters, we describe the details of a formalism that can express all these things, as well as formal links between the ecological and simulation modelling levels. In this chapter we concentrate on the theory. In chapter 6 we give important details on how the formalism is implemented. In chapter 7, we describe the how ELK may be used to interactively construct statements in this formalism which collectively constitute descriptions of ecological knowledge, sys-

tems, and simulation models. The majority of the formalism is implemented; we note exceptions.

To test our knowledge ontology hypotheses, we use these four levels as the basis for presenting the formalism (they will usually correspond to section headings).

We begin by clarifying some more terminology. After that we discuss our use of the typed lambda calculus generally and give the details of the key features in ElkLogic. Using the same Serengeti example, we then illustrate how ElkLogic is used to represent the above predications. This covers the major features of our formalism. A summary of the details of ElkLogic is found in appendix C.

5.2 Introducing ElkLogic

We describe an augmented version of the representation described in § 2.3.3 which meets our requirements. We call it *ElkLogic*; it embodies virtually everything in EcoLogic [Bundy & Uschold, 1989] and extends the latter in important ways. ElkLogic (also referred to as 'our formalism', or 'the formalism') consists of a collection of constructs which can express the diverse set of predications listed at the beginning of § 5.1. That list serves as a concise summary of the intended semantics of our formalism. Operationally, (*i.e.* from an implementation point of view) the semantics of each construct is embodied in how the system uses it. Some are used to ensure ecological consistency, others are used to guide the dialogue, etc. We effectively have separate sub-languages for each of the four layers in our ontology. This is because each kind of predication is

- classified into one of the four layers,
- and has a corresponding construct.

Elklogic also contains explicit bridging constructs which record various idealisations as well as the ecological meaning of model variables and parameters. For example, in our representation, the fact that the model variable *n_wb* simulates the attribute *number* of the entity *wb_pop* is explicitly recorded, as well as the fact that the integer valued attribute *number* is idealised as being positive reals.

Because of the diversity of the required predications, there is no obvious way that everything could be captured in a single object-level language. However, both the object-level and meta-level language represent information from more than one

layer in the ontology. That is, the object/meta distinction is separate from those in the ontology.

5.2.1 Terminology

The terms 'specification', 'construct', 'language', and 'formalism' are used in many different ways both technically and otherwise. This is how we use these terms:

formalism: a set of syntactic and semantic conventions for describing something. (subsumes 'language')

construct: a representation primitive together with its type. These define the syntax of a formalism. In our context this will usually be some logical term.

instantiated construct: an instantiation of a single construct of a formalism.

specification: an instantiation of one or more constructs in a formalism.

Most of the time 'specification' will refer to a single instantiated construct. It can of course refer to a complete specification. Although it is common to use 'formalism' and 'language' as synonyms, we shall not use these terms interchangeably. We use the term 'formalism' in a more general sense; it may be composed of conceptually and/or formally distinct sub-languages. So, we will refer to Elk-Logic as a formalism, not a language. It includes all the meta- and object-level constructs that are use to characterise knowledge/information in all of the four levels in our ontology. These constructs collectively define various sub-languages which talk about different kinds of things, or about the same kinds of thing but from a different point of view.

5.2.2 Using the Typed Lambda Calculus

The formalism is based on order-sorted typed lambda calculus [Barendregt, 1985], and is similar to that described in [Cardelli, 1989]. It is important to note that we use this primarily as a representation framework. Because we have not found a need for it, we *make very little use of the proof theory* that comes with the lambda calculus. In particular, we do not require full unification or general theorem proving; nor are we directly concerned with soundness or completeness. We do however require evaluation (*e.g.* beta-reduction), and one-way matching (matching is discussed in § 7.5.2.2). Our use of the typed lambda-calculus is novel. Rather than use a rich type structure to constrain the search space for proofs [Cohn, 1985] we

use it to identify and constrain the search in the process of creating ecological models. The way that we use it facilitates ensuring consistency, model comprehension and many other benefits as discussed in chapter 4. The fact that we make ample use of a rich type structure, but largely ignore the proof theory is unusual[1].

The lack of proof-theoretic concerns, means that the usual problems of computational efficiency do not arise. This allows relatively unconstrained introduction and use of higher-order functions and meta-level constructs of various kinds to get the expressive power we find useful. Although we have a substantial kernel, we are still discovering new concepts that need to be represented, and devising new better ways to represent these and other already identified concepts. Thus, a rigorous formal account is neither necessary nor appropriate at this time. Instead, we give an informal presentation of the theory underlying the design and implementation of ELK. In doing this, we aim to:

- explain unambiguously what ELK achieves and how
- lay the foundation for a rigorous formal account in the future.

Lest the reader think our job has been unduly simplied, it is important to realise that instead of formal rigour, we are required to provide an interface for our logic formalism. It must enable ecologists who are neither logicians, mathematicians, nor computer scientists to describe ecological knowledge, systems, and models which are represented in a logic formalism.

5.3 Types, Sorts, and Sets

We begin with an overview of the object language which is used to describe some general/ecological information, all of the description of the ecological system, and all of the runnable model. We introduce meta-level constructs as required. A key feature of our formalism is the rich type structure, particularly with respect to sets. The most important primitive concepts on which ElkLogic is founded are:

1. sorts
2. set membership

---

[1] Tony Cohn, personal communication

3. basic subsort relation

The base types are called *sorts*. For example, *real* is the sort of real numbers and *sheep* is the sort of sheep. We use the symbol $\mathcal{S}$ to denote the set of all sorts.

We use the notation $X:T$ to denote that the type of X is T. This usually means that the thing denoted by 'X' is a member of the set denoted by 'T'. For example: $1.4{:}real$.

$S_1 \sqsubset^s_\circ S_2$ says that S_1 is a *basic subsort* of S_2. This relation is neither reflexive, symmetric, nor transitive. Formally:

$$\begin{aligned}
&\forall S_1, S_2, S_3{:}\mathcal{S}.\\
&S_1 \not\sqsubset^s_\circ S_1\\
&S_1 \sqsubset^s_\circ S_2 \rightarrow S_2 \not\sqsubset^s_\circ S_1\\
&(S_1 \sqsubset^s_\circ S_2 \sqsubset^s_\circ S_3) \wedge (S_1 \sqsubset^s_\circ S_3) \quad \rightarrow \quad (S_1 = S_2) \vee (S_2 = S_3)
\end{aligned}$$

This defines a tree of sorts. The root node is *indiv*, most general sort. It is useful to define $\sqsubset^s$ (proper subsort) and $\sqsubseteq^s$ (subsort) in terms of $\sqsubset^s_\circ$. $\sqsubset^s$ is the transitive closure of $\sqsubset^s_\circ$, and is irreflexive. $\sqsubseteq^s$ is the reflexive, transitive version. Formally:

$$\begin{aligned}
&\forall S_1, S_2, S_3{:}\mathcal{S}.\\
&\qquad S_1 \sqsubset^s_\circ S_3 \quad \rightarrow \quad S_1 \sqsubset^s S_3\\
&S_1 \sqsubset^s_\circ S_2 \wedge S_2 \sqsubset^s S_3 \quad \rightarrow \quad S_1 \sqsubset^s S_3 \qquad (5.1)\\
&\qquad S_1 \sqsubset^s S_2 \quad \rightarrow \quad S_1 \sqsubseteq^s S_2\\
&\qquad\qquad\qquad\quad S_1 \sqsubseteq^s S_1 \qquad (5.2)
\end{aligned}$$

From this sort hierarchy, in conjunction with various type constructors described below, we shall define a subtype relation '$\sqsubseteq$'. To avoid Russel's paradox, there is no most general type of which everything is an instance, and every type is a subtype. We use S for variables ranging over sorts, and T for those ranging over types.

Before we give the details of the type system, we discuss the nature and use of sets. These are of chief importance in ElkLogic. We adopt the usual semantics for sorts: a sort denotes the set of entities of that sort. Thus, '*sheep*' denotes the set of all sheep. In most logics with subtypes (*e.g.* [Cardelli, 1989]), if a set is not a sort, it is a subtype defined in terms of the properties of its members. For example, the even numbers might be a subtype defined as follows:$\{X|\exists N{:}integer.\ 2 \cdot N = X\}$. If the set is arbitrary, (*e.g.* $\{1, 5, 11\}$) then a kind of degenerate property may be

used to define it:

$$\{X|X = 1 \vee X = 5 \vee X = 11\}$$

In modelling ecological systems there is a need to represent sets, (*e.g.* $flk1$ and $flk2$ might denote two flocks of sheep). The usual way to proceed is to define these flocks as subtypes. However, there is no obvious way to do this because:

- There may be no common property or set of properties *of the sheep* that distinguishes the different flocks of sheep. For example, the reason for distinguishing them may be because one flock receives higher grade food, or is less subject to predators.
- We cannot use the degenerate property based on a disjunction of equality with respect to the members, because frequently *the members are of no interest, and will not be explicitly defined.*

There are two additional arguments against using subtypes for representing the flocks of sheep:

- it is counterintuitive
- the reasons for distinguishing the flocks may change

To say that the two flocks are different subtypes, suggests that in some sense the members are different kinds of sheep. What kinds of sheep are the members of our two example flocks? We could say the sheep in the first flock are 'sheep that are fed high grade food' which one might argue is a kind of sheep just as 'black sheep' is a kind of sheep. However, this is quite unnatural. Intuitively, the sheep in each set are the *same* type, they have the same attributes.

If a user decides that they are not interested in food, but something else the definition of the flocks in terms of attributes would require changing. This is undesirable and should be unnecessary.

So for various reasons, using subtypes for representing these flocks is problematic. In seeking an alternate approach we note the following requirement. The attributes that the two flocks have depends on the sort of the members (*i.e. sheep*). For example, because sheep have the attribute biomass, sets of sheep have the attributes 'average biomass', 'total biomass', etc. So, the representation for the type of these flocks should in some way be defined in terms of the sort *sheep.*

We introduce the type constructor *set* for representing the type of a set of entities of a given sort (or type). For example, *set*(*sheep*) is the type of sets of

sheep. Instead of representing sets of sheep as subtypes, we represent them as *instances* of this induced set type (*i.e.* $flk1{:}set(sheep)$ and $flk2{:}set(sheep)$).

Note that $set(T)$ is analogous to $list(T)$ in many type theory systems. There are two important differences:

- order is not relevant
- there is no requirement that the members are *in principle* able to be identified.

Before we proceed, we introduce some notation for typing functions and relations defined on types. We use the power operator [Cardelli, 1988]. For an arbitrary type T, '$\mathcal{P}(T)$' denotes the power type of T which is defined to be the type of all subtypes of T, (including T). Thus, $T : \mathcal{P}(T)$, and if $T_1 \sqsubset T_2$ then $T_1{:}\mathcal{P}(T_2)$.

We now give the rules for creating types from sorts.

1. every sort is a type
   If $S \sqsubseteq^s indiv$ then S is a type
2. every type induces a set type which is the type of sets of things of that type.
   If T is a type$^{(*)2}$ then $set(T)$ is a type
3. every type induces a power type which is the type of subtypes of that type.
   If T is a type$^{(*)}$, then $\mathcal{P}(T)$ is a type.
4. types may be combined using a type union operator: $\sqcup$ (see 5.11)
   If T_1 and T_2 are types$^{(*)}$ then $T_1 \sqcup T_2$ is a type
5. types may be combined using a type complement operator $\setminus_t$ (see 5.12)
   If T_1 and T_2 are types$^{(*)}$ and $T_2 \sqsubseteq T_1$ then $T_1 \setminus_t T_2$ is a type
6. tuples are types
   If $T_1, T_2, \ldots, T_n$ are types, then $T_1 \times T_2 \times \ldots \times T_n$ is a type
7. mappings are types
   If $T, T_1, T_2, \ldots T_n$ are types, then $T_1 \times T_2 \times \ldots \times T_n \mapsto T$ is a type

We sometimes use the notation T^n as an abbreviation for $T \times \ldots \times T$ for n occurrences of T. The types of the subsort relations are given in 5.3.

$$\sqsubset_o^s, \sqsubset^s, \sqsubseteq^s : \mathcal{P}(indiv) \times \mathcal{P}(indiv) \mapsto bool \tag{5.3}$$

---

[2] The $*$ indicates that the type must not require rules 6 or 7 to be formed.

We will induce a hierarchy of types where $\sqsubset$ is the general case of $\sqsubset^s$. $\sqsubset$ is transitive, but not reflexive. The reflexive version is $\sqsubseteq$. Formally:[3]

$$\begin{aligned} &\text{If } T_1, T_2, T_3 \text{ are types then} \\ &\forall S_1, S_2 \sqsubset^s indiv. S_1 \sqsubset^s S_2 \;\rightarrow\; S_1 \sqsubset S_2 \\ &T_1 \sqsubset T_2 \wedge T_2 \sqsubset T_3 \;\rightarrow\; T_1 \sqsubset T_3 \qquad (5.4) \\ &T_1 \sqsubset T_2 \;\rightarrow\; T_1 \sqsubseteq T_2 \\ &T \sqsubseteq T \qquad (5.5) \end{aligned}$$

We distinguish three main kinds of types:

1. *entity types:* any type that can be constructed using only rules 1, 2, 4, and 5. The most general type of entity is *entity*.
   e.g. 1:*real*
2. *relation types:* any type constructed using rule 7 last where $T = bool$.
   e.g. $\sqsubset$: $\mathcal{P}(entity) \times \mathcal{P}(entity) \mapsto bool$
3. *function types:* any type constructed using rule 7 last where $T \neq bool$.
   e.g. $n\_wb : year \mapsto real$

We are mostly concerned with the entity hierarchy. Our use of function and relation types is limited to one-off typing of functions and relations. We do no type inference except for entity types. This is why, we restrict the application of the type formation operators $set, \mathcal{P}, \sqcup$, and $\setminus_t$ to exclude types that require rules 6 and/or 7 to be formed.

Just as sorts denote the set of entities of a certain sort, types denote the set of all entities of a certain type. Entities are members of the set denoted by their type. For example, if $ln : lion$ then 'ln' denotes a member of the set denoted by '*lion*'. Similarly, if $ln\_pop : set(lion)$ then ln_pop denotes a member of the set of sets of lions. In general, if $E : set(T)$, then 'E' denotes a subset [*not* a member] of the set denoted by T. It follows that for any type T, the set denoted by '$set(T)$' is the power set of the set denoted by 'T'. However, $set(T)$ and $\mathcal{P}(T)$ are not the same because in general, sets are not types. For example, if $lion : \mathcal{S}$, $ln1, ln2 : lion$, then $\{ln1, ln2\} : set(lion)$, but it is *not* the case that

[3] In this and other similar definitions, we use the closed world assumption.

$\{ln1, ln2\}{:}\mathcal{P}(lion)$, nor is $ln1{:}\{ln1, ln2\}$. To denote that $ln1$ is a member of the set, we write $ln1 \in \{ln1, ln2\}$.

Except in one special case, if $E{:}set(T)$ then E is an entity, *not* a type. The special case is when E corresponds to the entire set denoted by the type. For example because '*lion*' denotes the set of all lions, it follows that $lion{:}set(lion)$.

Note also that we do not allow formation of types using set abstraction. This, is related to the fact that not all sets are types, and differs from many logics using subtypes (*e.g.* [Cardelli, 1989]).

Part of $\sqsubset$ coincides with the sort hierarchy. Another major part of it is is derived from the following rule:

$$\forall T_1, T_2 \sqsubset entity. T_1 \sqsubset (T_1 \sqcup T_2) \wedge T_2 \sqsubset (T_1 \sqcup T_2) \tag{5.6}$$

The relationship between the type hierarchy and instances is given in (5.7) and (5.8). This says that an entity inherits all the supertypes of its type. For example, if $ln{:}lion$ then ln inherits the types *mammals*, *animals*, etc. from *lion*.

$$\forall E. \forall T_1, T_2 \sqsubseteq entity.$$

$$T_1 = T_2 \rightarrow E{:}T_1 \leftrightarrow E{:}T_2 \tag{5.7}$$

$$T_1 \sqsubset T_2 \rightarrow E{:}T_1 \rightarrow E{:}T_2 \tag{5.8}$$

This means entities do not have unique types. For primitive entities (*i.e.* whose types are sorts, not set types) we use the notation $E{:}_\circ S$ to denote that the least (*i.e.* most specific) type of entity E is S. It is unique because the sort hierarchy is a tree. Formally,

$$\forall E. \forall T_1 \sqsubseteq entity.$$

$$E{:}_\circ T_1 \rightarrow E{:}T_1$$

$$E{:}_\circ T_1 \leftrightarrow \forall T_2 \sqsubseteq entity.(E{:}T_2 \wedge T_2 \sqsubseteq T_1) \rightarrow T_2 = T_1 \tag{5.9}$$

The properties of type complement and union are given in 5.11 and 5.12. Associativity and commutativity for $\sqcup$ are inherited from logical disjunction. It is important to note that we use the union and complement functions primarily for typing. With one exception (to be noted later), we *do not allow* instances whose least types can only be expressed using $\sqcup$ and/or $\setminus_t$. For instance, ELK provides no facility for creating the hypothetical instance: $some\_cat{:}lion \sqcup tiger$. We do

this in the interest of simplicity; it is forbidden in [Cardelli, 1989] for reasons of computational tractability.

$$\sqcup, \setminus_t : \mathcal{P}(entity) \times \mathcal{P}(entity) \mapsto \mathcal{P}(entity) \tag{5.10}$$

$$\forall E{:}entity.\forall T_1, T_2 \sqsubseteq entity.E{:}(T_1 \sqcup T_2) \leftrightarrow E{:}T_1 \vee E{:}T_2 \tag{5.11}$$

$$\forall E{:}entity.\forall T_1, T_2 \sqsubseteq entity.E{:}(T_1 \setminus_t T_2) \leftrightarrow E{:}T_1 \wedge \neg E{:}T_2 \tag{5.12}$$

We need rule 5.13 to be able to deduce that $set(animal) \sqsubseteq set(lifeform)$, and $set(set(animal)) \sqsubseteq set(set(lifeform))$. This induces infinitely many parallel hierarchies directly analogous to the one for sorts (and thus constitutes another part of the definition of $\sqsubset$). The root nodes of the induced hierarchies are $set(indiv)$, $set(set(indiv))$, etc., where $indiv$ is the most general sort.

$$T_1 \sqsubset T_2 \rightarrow set(T_1) \sqsubset set(T_2) \tag{5.13}$$

We now define *entity*, the most general entity type. It is an induced type, not a sort. It is a supertype of every type that can be constructed from sorts using *set*, $\sqcup$, and $\setminus_t$ (not $\mathcal{P}$). We only have two kinds of entities, primitive ones whose types are sorts, and sets. Most of the time it is not necessary to distinguish between simple sets, and sets of sets of entities of a certain sort; sometimes trying to maintain the distinction causes problems (we give an example later). To blur this distinction, we use the notation $\#S$ to refer to the type of arbitrarily nested sets whose flattened versions contain members all of whose types are S. For instance, $\{1, \{2\}\}{:}set(integer \sqcup set(integer))$. The type of the flattened version is $set(integer)$. The # notation captures all types of sets for a particular sort, so it follows that $entity = indiv \sqcup \#indiv$. For any sort S, we use the notation $S^{\odot}$ to refer to the type $S \sqcup \#S$. Thus, $entity = indiv^{\odot}$. In figure 5–1 we give the formal definition of # and $^{\odot}$, with examples.

Using a simple recursive algorithm, ELK can infer the more specific type information for a set entity from its substructure. This will become more clear when we introduce our representation for substructure. We shall rarely be concerned with such complex sets, but since they might be needed, we do not prohibit them. If such complex set substructure were to be defined, we would never need to know it's exact details. The # notation allows us to blur the distinction between simple sets or ones with complex substructure. A pride of lions is still a pride of lions whatever its substructure.

$$\#,\ ^{\odot}, set^{(i)} \ :\ \mathcal{P}(indiv) \mapsto \mathcal{P}(entity)$$

$$\begin{array}{lcll} \forall S \sqsubset indiv. & & & \\ set^{(0)}(S) & = & S & \\ set^{(i)}(S) & = & set(set^{(i-1)}(S)) & for\ i \geq 1 \end{array}$$

$$\begin{array}{rcl} S^{(0)} & = & S \\ S^{(n+1)} & = & S^{(n)} \sqcup set(S^{(n)}) \\ S^{\odot} & = & \bigsqcup_{i=0,\infty} S^{(i)} \\ \#S & = & S^{\odot} \setminus_t S \end{array}$$

From these definitions and from rule 5.6 (page 138) it follows that:

$$\begin{array}{l} \forall i,j{:}integer.\forall S \sqsubset indiv.\forall T \sqsubseteq entity. \\ \qquad \begin{array}{lcl} i \leq j & \rightarrow & set^{(i)}(S) \sqsubset set(\#S) \sqsubset \#S \sqsubset S^{\odot} \\ S^{\odot} & = & S \sqcup \#S \\ set(T) & \sqsubseteq & \#indiv \\ set(S^{\odot}) & = & \#S \end{array} \end{array}$$

For example:

$$\begin{array}{rcl} \{\{1,2\},\{3,4\}\} & : & set^{(2)}(integer) = set(set(integer)) \\ \{1,2,\{3,4\}\} & : & set(integer^{(1)}) = set(integer \sqcup set(integer)) \\ \{\{1,2,\{3,4\}\},5\} & : & set(integer \sqcup set(integer^{(1)})) \\ & & = set(integer, set(integer \sqcup set(integer))) \end{array}$$

Figure 5–1: Formally defining collections

As an example of where we need to know something about the details of the substructure, suppose we are interested in computing the annual maximum average weight of a pride of lions ($ln\_pop : \#lion$). This could be interpreted in various ways. It could mean taking the average for the pride at 12 monthly intervals, and then taking the maximum of the 12 results. Alternatively it could mean an average is computed for (say 5) different sub-prides in the overall pride, and then taking the maximum of those 5 results. In the former case, the most specific sort of *ln_pop* may be $set(lion)$, however for the second case, it must be a subtype of $set(\#lion))$ (*e.g.* $set(set(lion)))$. It does not have to be $set(set(lion))$ because, there may be further subdivisions within the pride which can be ignored for the purpose of the computation (*i.e.* flattened).

This is why we never need to know the exact details of a set entity's substructure (nor therefore its most specific type). In the specification process, users never have to decide *a priori* anything about what its substructure might eventually be. In ELK, set entities always are explicitly typed $\#S$ for some sort S. This is the exception to the rule about not allowing instances whose most specific types require $\sqcup$ and/or $\setminus_t$ to be expressed. For the remainder of this thesis, we shall always refer to set types using # rather than *set* unless it is necessary to make the distinction. Entities that are typed using # will usually be referred to as *collections*, rather than set entities.

The $^{\odot}$ notation is convenient when we do not wish to commit to whether we have a set or not, but we know what sort we are talking about. We will use this notation for typing.

There are three main subsorts of *indiv*, corresponding to the entity ontology discussed in chapter 2. *ecol_indiv* is the most general sort of ecological entity excluding collections. Its main subsorts include *phys_obj*, *plant*, *animal*, etc. For the example model we also need more specialised sorts for lions and wildebeest (*lion*, *wb*). There are also sorts of parts (such as *branch*, *leg*, *tail*) of composite entities. *value* is the sort of values of ecological entities. This includes real, positive, and natural numbers, colours, male/female etc. Finally, we have a few sorts of time entities: *time*, *year*, *day*, etc. The gross structure of the type hierarchy is shown in figure 5–2. Some of the details not included in the figure are given below:

$$natural \sqsubset^s positive \sqsubset^s real \sqsubset^s value$$
$$year, month, day \sqsubset^s time$$

Constraints

In the implementation, we have the following constraints:

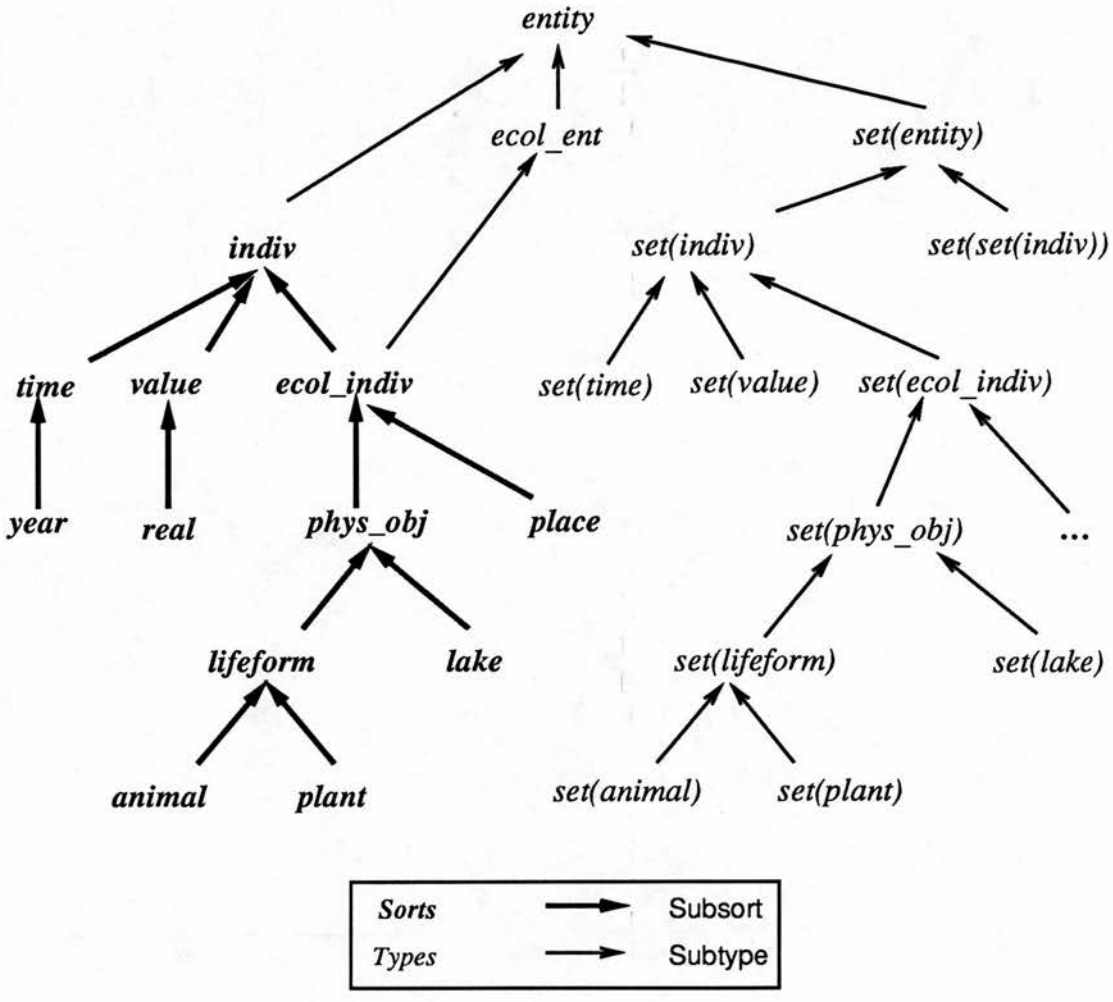
- There are no entities E such that $E:_\circ indiv$.
- There are no entities E such that $E:_\circ value$.
- No entity may have two different types unless one is a subtype of the other. Formally:
  $E{:}T_1 \wedge E{:}T_2 \rightarrow T_1 \sqsubseteq^s T_2 \vee T_2 \sqsubseteq^s T_1$
- All sets are created with types of the form $\#S$ for some sort S. More specific types expressed in terms of *set* and $\sqcup$ are inferred as necessary.
- Except indirectly using $\#S$, there are no entities whose types require the use of $\sqcup$ or $\setminus_t$ to be expressed.

5.4 Attributes, Variables, Processes

In § 2.5.5 we said that attributes apply to entities and have values. For example, the attribute 'colour' applies to physical objects and has values 'red', 'green', etc. Attributes are naturally represented as functions from ecological entities and time to values. We represent all proper model variables as functions from time to values, and parameters simply as values. This agrees with the types given in table 2–1 for the variables in the example model. Processes are represented by relations and functions. For example:

Concept	Example	Type
attribute:	$number:$	$\#indiv \times time \mapsto natural$
variable:	$n\_wb:$	$time \mapsto natural$
parameter:	$wb\_fec:$	$real$
process:	$predation:$	$animal^{\odot} \times animal^{\odot} \mapsto bool$
effect:	$wb\_eaten':$	$time \mapsto positive$
partial rate variable:	$wb\_eaten:$	$time \mapsto positive$

By *number* we mean the number of members in the flattened set, not the literal set. The sort hierarchy gives us the attribute inheritance that we require. Thus weight



entity	the most general type of entity
indiv	the most general type of primitive entity
set(entity)	the most general type of set entity ($= \#indiv$)
ecol_ent	the most general type of ecological entity ($= ecol\_indiv^{\odot}$)
ecol_indiv	the most general type of primitive ecological entity

Note that $\sqsubset$ is a partial order, not a tree.

Figure 5–2: Part of the Subtype Relation

also applies to *animal* because $animal \sqsubset^s phys\_obj$. The relation *predation* represents all the animals and/or animal populations that are preying on one another. If *predators*:#*animal*, and *wb_pop*:#*wb*, then *predation*(*predators*, *wb_pop*) says that the population denoted by '*predators*' is preying on the population denoted by '*wb_pop*'. '*wb_eaten*'' is a function representing one effect of the predation process which is to reduce the number of wildebeest. It is the rate at which wildebeest are being eaten by the predators at any given time in the real or hypothetical ecological system being modelled. *wb_eaten* here is the same variable in the example set of equations in chapter two. It is the idealised version of *wb_eaten'* here which is the function corresponding to the 'real' effect. This is analogous to model variables like *n_wb* idealising the functions corresponding to 'real' attributes (*e.g. number* of *wb_pop*). When referring to model variables and the effects or attributes they are idealising, the non-primed ones are the idealisations of the primed ones. This is not the only use of primes, however.

5.5 Higher Order Functions; Induced Attributes

We shall have occasion to use higher-order functions in a number of ways. The most obvious case arises from the need to represent differential equations. We define a second order function, *rate* which maps one unary first order function to another of the same type. Formally:[4]

$$\forall T \sqsubseteq entity. V \sqsubseteq value.\ \ rate:\ \ (T \mapsto V) \mapsto (T \mapsto V)$$

For our purposes, T and V will almost always be *time*, and *real* but could in principle be any types whose instances were totally ordered. Thus, *rate* is polymorphic across different sorts. N.B. Elklogic currently does not represent the concept of orderedness, although ELK knows about it via special-purpose code.

We also use second order functions to represent concepts like maximum and average. These take two arguments. The first is a unary function defined on some type of entity; the second is a set of entities of the same sort. It returns a value from the range of the unary function. Other concepts of exactly the same type

[4] The idea of using higher-order functions for representing rate, average, maximum, etc. was Alan Bundy's.

include *total*, *median*, *mode*, and possibly others. *average* is arithmetic mean. Formally:

$$\forall T \sqsubseteq entity. \forall V \sqsubset value. maximum : (T \mapsto V) \times set(T) \mapsto V \qquad (5.14)$$

Although literally, these functions produce values, we use them in conjunction with λ-abstraction to produce new functions. For example, from the function n_wb, we induce the function max_n_wb which is the maximum value of n_wb for a set of times. For example:

$$\begin{aligned} max\_n\_wb &= \lambda P : set(time).maximum(n\_wb, P) \\ max\_n\_wb(80s) &= maximum(n\_wb, 80s) \\ max\_n\_wb &: set(time) \mapsto value \end{aligned}$$

The second line gives two alternate ways to represent the maximum number of the wildebeest population in the 80s, where $80s : set(year)$. The right hand side is the preferred option, and is supported by Elklogic. This avoids the need for proliferation of primitives like max_n_wb, should the concept of maximum need to be used over and over.

We can induce new attributes in the same way. Below we show how to represent the maximum weight of a physical object over some time period ($max\_wt(wb1, 80s)$) as well as the maximum weight of a set of physical objects at a single time ($max\_wt'(wb\_pop, yr80)$).

$$\begin{aligned} &wb\_pop = \{wb1, wb2\};\ wb1 \in wb\_pop;\ wb2 \in wb\_pop \\ &max\_wt = \lambda O{:}phys\_obj.\lambda P{:}set(time).maximum(\lambda T{:}time.weight(O,T), P) \\ &max\_wt(wb1, 80s) = maximum(\lambda T{:}time.weight(wb1, T), 80s) \\ &max\_wt : phys\_obj \times set(time) \mapsto value \end{aligned}$$

$$\begin{aligned} &max\_wt'(wb\_pop, yr80) = maximum(\lambda O{:}phys\_obj.weight(O, yr80), wb\_pop) \\ &max\_wt' = \lambda Os{:}set(phys\_obj).\lambda T{:}time. \\ &\qquad maximum(\lambda O{:}phys\_obj.weight(O, yr80), Os) \\ &max\_wt' : set(phys\_obj) \times time \mapsto value \end{aligned}$$

The names max_wt and max_wt' are concise, but do not capture the relationship between the two functions; the full representation of the functions retains the connection, but is verbose and hard to read. Thus, we introduce a more concise notation which retains the appropriate connections. The difference between

max_wt and *max_wt'* is that the former is a maximum taken over a set of times, and the latter over a set of ecological entities. We use the third order functions *qnam_tim* and *qnam_ent* to represent the binary functions derived using *average*, *maximum*, etc. The *qnam_* terms may be viewed as structured names for the induced (or qualified) functions (hence the prefix *qnam_*). For example:

$$
\begin{aligned}
&qnam\_tim(maximum, weight) && = \ max\_wt \\
&qnam\_ent(maximum, weight) && = \ max\_wt' \\
&qnam\_tim(maximum, weight)(wb1, 80s) && = \\
&\qquad maximum(\lambda T{:}time.weight(wb1, T), 80s) \\
&qnam\_ent(maximum, weight)(wb\_pop, yr80) && = \\
&\qquad maximum(\lambda O{:}phys\_obj.weight(O, yr80), wb\_pop)
\end{aligned}
$$

These may be nested. The following expressions represent the maximum average weight of the wildebeest population in the 80s.

$$
\begin{aligned}
&maximum(\lambda Y{:}year.average(\lambda O{:}phys\_obj.weight(O, Y), wb\_pop), 80s) \\
&\qquad qnam\_tim(maximum, qnam\_ent(average, weight))(wb\_pop, 80s) \\
&\qquad\qquad qnam\_tim(maximum, qnam\_ent(average, weight)) : \\
&\qquad\qquad\qquad set(phys\_obj) \times set(time) \mapsto positive
\end{aligned}
$$

For proper variables which are unary functions we use *qnam*. For example,

$$qnam(maximum, n\_wb) = max\_n\_wb$$

The *qnam_* constructs are used only as a more concise notation; they give us nothing new in expressive power. Their types and how they are used are derived directly from the type of *maximum*, *average*, etc. They are sometimes easier to understand and manipulate than the equivalent λ expressions. To type these constructs, note that the first argument must always be one of *maximum*, *average*, etc. These are all of the same type (see 5.14). The second argument is a binary function from entities and times to values (*e.g. weight*). Formally:

$$\forall T \sqsubseteq entity.\forall V \sqsubset value.\forall T_m \sqsubseteq time.$$

$$\begin{aligned}
qnam_tim : ((T \mapsto V) \times set(T) \mapsto V) &\times (T \times T_m \mapsto V) \\
&\mapsto (T \times set(T_m) \mapsto V) \\
qnam_tim(Q, F) &= \lambda X, Y.Q(\lambda T{:}time.F(X, T), Y)
\end{aligned}$$

$$\begin{aligned}
qnam_ent : ((T \mapsto V) \times set(T) \mapsto V) &\times (T \times T_m \mapsto V) \\
&\mapsto (set(T) \times T_m \mapsto V) \\
qnam_ent(Q, F) &= \lambda X, Y.Q(\lambda E{:}entity.F(E, X), Y)
\end{aligned}$$

$$\begin{aligned}
qnam : ((T \mapsto V) \times set(T) \mapsto V) &\times (T \mapsto V) \\
&\mapsto (set(T) \mapsto V) \\
qnam(Q, F) &= \lambda X.Q(\lambda E{:}entity.F(E), X)
\end{aligned}$$

Meta-Level Types

We do not capture everything we need to know about rate, average, maximum in the typing. This includes order information, and whether addition is defined on the relevant types. *maximum*, *minimum*, *median*, and *mode* require that the values are ordered. *total* further requires that addition is defined; *average* requires all this plus that division is defined. To capture this formally, we would require subtypes of the polymorphic type $(T \mapsto V) \times set(T) \mapsto V$ which made the required distinction. Such mechanisms are present in the formalism described in [O'Keefe, 1985], however Elklogic is not up to this task. Instead we use a separate, informal mechanism for making the required distinctions.

There are two kinds of second order functions both with the same object-level type. We use the meta-type *i.e. squal* to refer to this type (*e.g. average*:*squal*). The Elklogic type system is unable to make the distinction between *average* and *maximum* at the object-level, so we introduce the meta-level sub-type *ad_squal* of which *average* is a (meta-level) instance, but not *maximum*. ELK uses this meta-level information to treat *average* and *maximum* differently as required. At the object-level their types are indistinguishable.

Another case where meta-level types are useful is in distinguishing between attributes, effects, and variables. For example, the object-level types of effects and partial rate variables are identical ($time \mapsto value$). In chapter 6 we present various meta-level constructs which are used to make the distinction. Thus we can say *wb_eaten'*:$effect$ and *wb_eaten*:*variable* where $effect$ and *variable* are meta-

types. We also use the meta-type *attribute* for attributes (*e.g. number:attribute*). The main meta-level types are summarised below.

$$
\begin{array}{lcl}
\forall T \sqsubseteq entity. \forall V \sqsubset value. & & \\
F : squal & \rightarrow & F : (T \mapsto V) \times set(T) \mapsto V \\
F : variable & \rightarrow & F : (time \mapsto value) \sqcup value \\
F : propvar & \rightarrow & F : time \mapsto value \\
F : parameter & \rightarrow & F : value \\
F : attribute & \rightarrow & F : ecol\_ent \times time \mapsto value \\
F : process & \rightarrow & F : ecol\_ent^n \mapsto bool \\
F : effect & \rightarrow & F : time \mapsto value \\
 & & \\
qnam\_ent & : & squal \times attribute \mapsto attribute \\
qnam\_tim & : & squal \times attribute \mapsto attribute \\
qnam & : & squal \times propvar \mapsto propvar
\end{array}
$$

As well as being used to make distinctions that are not possible to represent in the object-level language, meta-types are also a convenient shorthand. They may function as type abbreviations. The latter use is illustrated for the *qnam_* constructs above. Although for *qnam_tim* and *qnam_ent* we lose information, it is a convenient abstraction. We also imply something that the full types miss, namely that we only apply them to attributes, not arbitrary binary functions. We only apply *qnam*, to variables, not attributes. We shall use the term *abstract type* to refer to types which are abstract, simplified versions of the full type; they are expressed using meta-types. It is important to remember that although these are simplified and may carry less object-level information, they also carry meta-level information that the full types miss. Thus, both are useful.

Another kind of meta-level types are used purely for the implementation, and have no object-level interpretation (eg *increase* or *decrease* as specifications of an effect of a process). We mention these here, but will not use them until chapter 6. These correspond to the slots that users fill in as part of the interface to ELK. Their role in the thesis is as program documentation. Figure C–3 gives a complete summary of the meta-level types used by ELK.

5.6 Substructure

5.6.1 Motivation

An important reason for defining substructure is to be able to refer to different components either singly or as a whole. This is achieved by bounded quantification. We have various kinds of quantification corresponding to the different kinds of sets we represent.

1. $\forall L{:}lion.foo(L)$
2. $\forall L \in \{lion1, lion2, lion3\}.foo(L)$
3. $\forall T \sqsubseteq entity.foo(T)$

The first kind is standard. The second is due to our use of sets as instances of set types rather than as types themselves. The third we have used extensively already.

5.6.2 The Basics

In chapter 2, we identified three kinds of component relation that can hold between entities: *member-set*, *subdivision-set*, and *part-composite*. As we blurred the details of the substructure of a set, while retaining the ability to make the distinctions when needed, we do the same for the three kinds of substructure relationships. In ElkLogic, we represent all substructure between entities using a single relation called component. We use the notation '$E_c \subset E_w$' to denote that the entity E_c is a proper component (*i.e.* not the same entity) of the [whole] entity E_w. This conforms to the notation used in [Bunt, 1986] which captured the same abstraction. $\subset$ subsumes the three relations: 'member of', 'subset of', and 'part of' as they are commonly used.

We have $\subset$ and $\subseteq$ defined in terms of $\subset_o$ in a manner similar to that for $\sqsubset^s$ and $\sqsubset$ in definitions (5.1) and (5.4). $\subset$ is also transitive. We omit the details.

If we wish to represent the fact that a particular paw is part of a particular lion which is in turn a member of a pride of lions which is a subdivision of a collection of prides which is in turn a subdivision of a predator population including other predator species, we do so as follows.

$$\subset_o, \subset, \subseteq: entity \times entity \mapsto bool$$
$$paw1{:}paw \quad lion1{:}lion \quad pride, ln\_pop{:}\#lion \quad predators{:}\#animal$$
$$paw1 \subset_o lion1 \subset_o pride \subset_o ln\_pop \subset_o predators$$

Using reflexivity and transitivity we can infer $paw1 \subseteq paw1$, $paw1 \subset ln\_pop$, etc. We can also infer which of the three kinds of component-whole relationship holds by analysing the types of the component and whole entities. For example, because $lion1{:}lion$ and $pride{:}\#lion$, the $lion1$ is a member of *pride*, not a subdivision, or part.

It should never happen that a lion population be a component of a paw, nor a paw to be a component of an integer. Less obviously, we do not wish to allow a collection of animals to be a component of a collection of lions. If the set of animals contained elephants, then it makes no sense to say it is component of a set of lions. If the set of animals does in fact contain only lions, then its type should be changed to $\#lion$ in which case the component relation makes sense. This encourages principled knowledge engineering.

To ensure consistency of this kind, we have a *possible component* relation: $\subset^p$. Its intended semantics is as follows: '$T_c \subset^p T_w$' denotes that [in the world] it makes sense for an entity of type T_c to be a component of an entity of type T_w. The way we use this relation is to constrain the use of $\subset_o$. Operationally, (*i.e.* from an implementation point of view) the semantics of $\subset^p$ is captured by the following rule:

$$\forall T_c, T_w \sqsubseteq entity. \forall E_c{:}T_c, E_w{:}T_w.\ T_c \not\subset^p T_w \ \rightarrow \ E_c \not\subset E_w \tag{5.15}$$

For the above examples, we require:

$$\#lion \not\subset^p paw \qquad paw \not\subset^p integer \qquad \#lion \not\subset^p \#animal$$
$$paw \subset^p lion \subset^p \#lion \subset^p \#lion \subset^p \#animal$$

Ideally, $\subset^p$ should:

1. permit every case of substructure that makes sense
2. prohibit every case of substructure that does not make sense

Because the nature of substructure is so complex, we do not succeed in defining $\subset^p$ to meet these requirements completely, however it is useful nevertheless. We further require that:

1. it should require a minimum of direct specification by the user, from which a large number of correct inferences may be drawn.
2. there should be no redundancy

Requirement 1 is necessary because it is infeasible to store all pairs in the $\subset^p$ relation explicitly. ELK uses a closed world assumption here assuming that $T_1 \not\subset^p T_2$ iff $T_1 \subset^p T_2$ is not derivable.

With respect to requirement 2, there should not be two ways to infer that something is a possible component of something else. Redundancy is both inelegant and gives rise to computational inefficiency. Furthermore, lack of redundancy facilitates requirement 1 above. One major technique for avoiding redundancy is through the use of transitivity. Thus, we shall define a relation $\subset^p_\circ$ whose transitive closure is $\subset^p$ in the usual way. However unlike $\sqsubset$ and $\subset$ we don't have an irreflexive version which we might call *proper possible component* because of the subdivision case. Using the informal sense of the word, it is "proper" to say that $\#lion \subset^p \#lion$.

Also unlike $\sqsubset^s_\circ$ and $\subset_\circ$ we do not define each link in the hierarchy explicitly. Instead both $\subset^p_\circ$ *and* $\subset^p$ are induced. The major reason for this is to avoid duplication. For example, we require $\#lion \subset^p \#animal \subset^p \#lifeform$, etc.; these mirror the sort hierarchy. In general, if $S_1 \sqsubset^s S_2$, then $\#S_1 \subset^p \#S_2$, thus the sort hierarchy is effectively embedded in $\subset^p$. In order to minimise or eliminate redundancy, we require that:

$$\begin{aligned}
&\forall T_1, T_3 \sqsubseteq entity \\
&\quad (T_1 \subset^p_\circ T_3 \;\leftrightarrow\; \exists S \sqsubset^s indiv. T_1 = T_3 = \#S (i.e.\ \text{subdivision case}) \\
&\qquad\qquad \vee\; \forall T_2 \sqsubseteq indiv. (T_1 \subset^p T_2 \subset^p T_3) \rightarrow T_2 = T_3) \qquad (5.16)
\end{aligned}$$

To infer $lion \subset^p \#lion \subset^p \#lion \subset^p \#animal$ we clearly need the following:

$$\forall S \sqsubset indiv. \quad S \subset^p \#S \qquad (5.17)$$

$$\forall S \sqsubset indiv. \quad \#S \subset^p \#S \qquad (5.18)$$

$$\forall S_1, S_2 \sqsubset indiv. \quad S_1 \sqsubset S_2 \quad \rightarrow \#S_1 \subset^p \#S_2 \qquad (5.19)$$

The sort hierarchy defines the set substructure of the primitive entities [in the world]. This does not enable us to infer $paw \subset^p lion$. We must also define the part-composite substructure. Separate from the sort hierarchy there is a part

hierarchy defined by the *possible part* relation: $\prec^p$. There is no need for a reflexive case. We define $\prec^p$ in terms of $\prec^p_\circ$ in the usual way:

$$\prec^p_\circ, \prec^p : \mathcal{P}(ecol\_indiv) \times \mathcal{P}(ecol\_indiv) \mapsto bool \tag{5.20}$$

$$\begin{aligned} \forall S_p, S_m, S_w \sqsubseteq ecol\_indiv. \\ S_p \prec^p_\circ S_w &\rightarrow S_p \prec^p S_w \\ S_p \prec^p_\circ S_m \wedge S_m \prec^p S_w &\rightarrow S_p \prec^p S_w \end{aligned} \tag{5.21}$$

Note that the part hierarchy is only defined on ecological entities. Although member and subdivision may be relevant for values, (*e.g.* $blue \subset \{red, green, blue\}$, $\{blue, red\} \subset \{red, green, blue\}$, we are not interested in composite entities that integers or colours may be parts of.

The intended semantics for this relation is analogous to $\subset^p$. '$S_p \prec^p S_w$' denotes that [in the world] all entities of sort S_w are composite entities which have parts of sort S_p. For example, '$claw \prec^p lion$' says that all sorts of lions have claws. $claw \not\prec^p animal$ says that "it is not the case that all sorts of animals have claws" which is true because only some animals have claws. Given this interpretation, if we were building a comprehensive knowledge base, we would have $claw \not\prec^p_\circ lion$, but $claw \prec^p_\circ feline$ from which we would infer that $claw \subset^p lion$ because $lion \sqsubset feline$. This suggests a relationship between the sort and part hierarchies. Finally, it should be possible for a set of claws to collectively be a component of a paw on the basis of the fact that a claw is part of a paw. To accommodate these cases, we also require:

$$\forall S \sqsubseteq ecol\_ent. \quad S_c \prec^p S_w \quad \rightarrow S_c \subset^p S_w \tag{5.22}$$

$$\forall T_p, T_w \sqsubset ecol\_ent. \quad T_p \prec^p_\circ T_w \quad \rightarrow \#T_p \subset^p T_w \tag{5.23}$$

$$\forall S_p, S_{w1}, S_w \sqsubseteq ecol\_ent. \quad (S_p \prec^p_\circ S_w) \wedge (S_{w1} \sqsubset S_w) \rightarrow S_p \subset^p S_{w1} \tag{5.24}$$

It turns out that (5.17), (5.18), (5.19) and (5.23) are basic cases and we may replace $\subset^p$ by $\subset^p_\circ$. However, (5.22) may be inferred from (5.17) and (5.23) and is thus redundant. For example $claw \subset^p_\circ \#claw \subset^p_\circ feline$ and (5.16) implies $claw \not\subset^p_\circ feline$, even though $claw \prec^p_\circ feline$. It turns out that (5.24) is also redundant and may be inferred from (5.17) and (5.25) the fifth and final basic case listed below. We need this to enable a set of claws to be a component of a lion, given a claw is part of a feline as in the example. Similarly, $claw \subset^p_\circ \#claw \subset^p_\circ lion$

and (5.16) imply *claw* $\not\subset^p_\circ$ *lion*.

$$\forall S_p, S, S_w \sqsubseteq \mathit{ecol\_ent}.\ \ (S_p \prec^p_\circ S_w) \wedge (S \sqsubseteq S_w) \ \rightarrow \#S_p \subset^p S \qquad (5.25)$$

Everything is now in place to define the possible component relation. Although $\subset^p_\circ$ and $\subset^p$ are similar in basic structure to the corresponding subtype, part and component hierarchies, $\subset^p_\circ$ is more complex than $\sqsubset^s_\circ$, $\prec^p_\circ$, or $\subset_\circ$. Rather than a single explicit relation which is directly defined, $\subset^p_\circ$ is defined in terms of 5 separate cases which are in turn based on the following three things:

1. the notion of a collection type; *i.e.* using # derived from *set*, $\sqcup$ and $\setminus_t$.
2. the sort hierarchy; *i.e.* $\sqsubset^s$
3. the possible part hierarchy; *i.e.* $\prec^p_\circ$

The first is described in § 5.3, and forms the backbone of the theory. The latter two are defined by the end user and/or knowledge-base builder. The five cases are represented by the following five relations, *pos_member, pos_subdiv, pos_coll_part, d_pos_subdiv,* and *d_pos_coll_part.* For each we give an English translation, and an example.

pos_member$(S, \#S)$ says that it is possible for an entity of type $\#S$ to be subdivided into member entities of type S.

e.g. *pos_member*$(\mathit{tree}, \#\mathit{tree})$ says that a collection of trees can be subdivided into member trees.

pos_subdiv$(\#S, \#S)$ says that is is possible for an entity of type $\#S$ to be a subdivided into entities of type $\#S$ (called subdivisions).

e.g. *pos_subdiv*$(\#\mathit{tree}, \#\mathit{tree})$ says that a collection of trees can be subdivided into (sub-)collections of trees.

pos_coll_part$(\#S_p, S_w)$ says that a composite entity of type S_w can subdivided into collections of parts of type S_p. This is true when $S_p \prec^p S_w$.

e.g. *pos_coll_part*$(\#\mathit{branch}, \mathit{tree})$ says that a tree can be subdivided into collections of branches.

d_pos_subdiv$(\#S_1, \#S_2)$ says that says that is is possible for an entity of type $\#S_2$ to be subdivided into entities of type $\#S_1$. This is true when $S_1 \sqsubset^s S_2$ (also called subdivisions).

e.g. *d_pos_subdiv*$(\#\mathit{pine}, \#\mathit{tree})$ says that a collection of trees can be subdivided into collections of pines (because a pine is a tree).

d_pos_coll_part$(\#S_p, S)$ says that a composite entity of type S can subdivided into collections of parts of type $\#S_p$. This is true when for some sort S_w, $S_p \prec^p S_w$ and $S \sqsubset^s S_w$

e.g. *d_pos_coll_part*$(\#branch, pine)$ says that a pine can be subdivided into collections of branches (because a pine is a tree, and a branch is a part of a tree).

Although there are 5 cases here, there are still only three fundamental kinds of component relation: *member-set*, *subdivision-set*, and *part-composite* (see beginning of § 5.6.2). The 2nd and 4th cases above both correspond to subdivisions-set; cases 3 and 5 both correspond to part-composite. Cases 2 and 3 differ from cases 4 and 5 in that the former are basic and the latter are derived (hence the *d_* prefix). The formal definition of $\subset^p_\circ$ and $\subset^p$ in terms of these five cases is given below. We use $\prec^p_\circ$ rather than $\prec^p$ to avoid redundancy which would arise because both $\prec^p$ and $\subset^p$ are transitive. Formally:

$$\forall S \sqsubset^s indiv. \qquad pos\_member(S, \#S) \tag{5.26}$$

$$\forall S \sqsubset^s indiv. \qquad pos\_subdiv(\#S, \#S) \tag{5.27}$$

$$\forall S_p, S_w \sqsubset^s ecol\_indiv. \quad (S_p \prec^p_\circ S_w) \leftrightarrow pos\_coll\_part(\#S_p, S_w) \tag{5.28}$$

$$\forall S_1, S_2 \sqsubset^s indiv. \quad (S_1 \sqsubset^s S_2) \leftrightarrow d\_pos\_subdiv(\#S_1, \#S_2) \tag{5.29}$$

$$\forall S_p, S, S_w \sqsubset^s ecol\_indiv. \quad S_p \prec^p_\circ S_w \wedge S \sqsubset^s S_w \leftrightarrow d\_pos\_coll\_part(\#S_p, S) \tag{5.30}$$

$$\begin{aligned} &\forall T_c, T_w \sqsubset^s indiv. \\ &pos\_member(T_c, T_w) \\ \vee\ &pos\_subdiv(T_c, T_w) \vee pos\_coll\_part(T_c, T_w) \\ \vee\ &d\_pos\_subdiv(T_c, T_w) \vee d\_pos\_coll\_part(T_c, T_w) \leftrightarrow T_c \subset^p_\circ T_w \end{aligned} \tag{5.31}$$

$$\begin{aligned} S_c \subset^p_\circ S_w &\rightarrow S_c \subset^p S_w \\ S_c \subset^p_\circ S_m \wedge S_m \subset^p S_w &\rightarrow S_c \subset^p S_w \end{aligned} \tag{5.32}$$

Thus $\subset^p_\circ$ is wholly induced from $\#$, $\sqsubset^s$, and $\prec^p_\circ$. All 5 basic cases use the notion of a collection. The first two use *only* this notion, the third uses the notion of a part, but not subsort. The fourth uses the notion of subsort, but not part, the final case uses both subsort and part.

$\subset^p$ in conjunction with $\subset$ not only enables us to blur the distinction between the various kinds of substructure, but it also enables us to recover the information if required. We merely analyse the types of the components. For example:

- If the type of the component is S and the type of the whole is $\#S$, then it is a member relation.
- If the types of the component and the whole are both $\#S$, it is a subdivision relation.

If $E_c \subset_\circ E_w$, where $E_c{:}T_c$ and $E_w{:}T_w$, then by axiom 5.15 $T_c \subset^p T_w$. This means a chain of one or more of the 5 basic cases defining $\subset^p_\circ$ holds. A trace of this path in the $\subset^p$ hierarchy gives the information we need. Suppose $paw \prec^p mammal$ and $lion \sqsubset mammal \sqsubset animal$; $paw1 \subset predators$ is permitted by $paw \subset^p \#animal$ which is derived by ordered application of *pos_member*, *d_pos_coll_part*, *pos_member* and *d_pos_subdiv*. Formally:

$$paw \subset^p_\circ \#paw \subset^p_\circ lion \subset^p_\circ \#lion \subset^p_\circ \#animal$$

Note that for ecological entities, $\in$ is a special case of the $\subset$ relation. Formally:

$$\forall E_m{:}T_m.\forall E_w{:}set(T_w).\quad E_m \subset E_w \wedge set(T_m) \sqsubseteq set(T_w) \quad \leftrightarrow \quad E_m \in E_w$$

$$e.g.\ \ paw1{:}paw \quad ln{:}lion \quad pride{:}set(lion) \quad predators{:}set(animal)$$

$$ln \subset pride \ \wedge \ set(lion) \sqsubseteq set(lion) \quad \leftrightarrow \quad ln \in pride$$

$$ln \subset predators \ \wedge \ set(lion) \sqsubseteq set(animal) \quad \leftrightarrow \quad ln \in predators$$

However, even though $paw1 \subset ln$, because there is no T_w such that $paw{:}set(T_w)$, $paw1 \notin ln$.

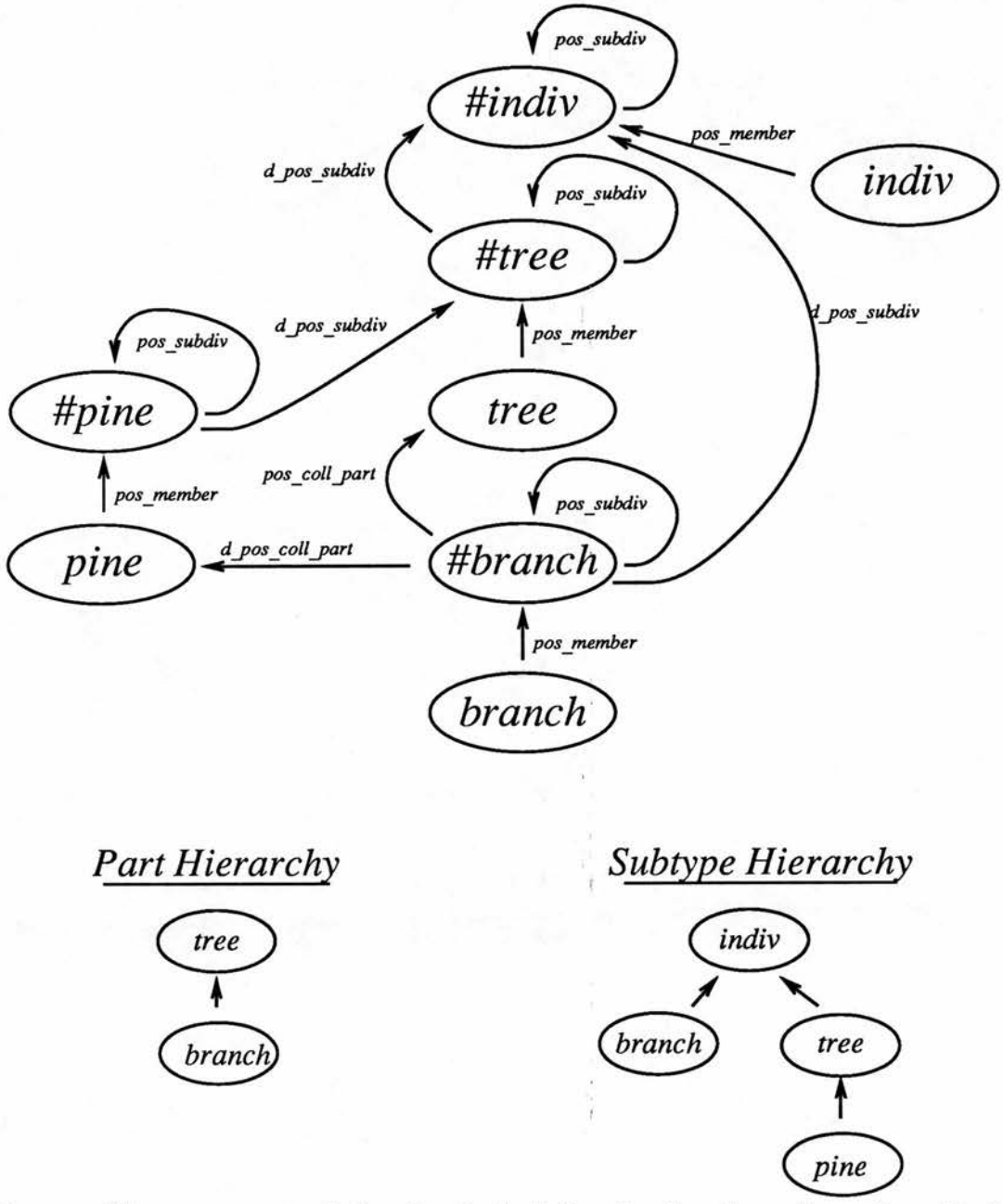
5.6.3 Example

We illustrate all 5 cases using the following trivial sort and part hierarchies.

$$branch \sqsubset^s ecol\_indiv \quad pine \sqsubset^s tree \sqsubset^s ecol\_indiv$$
$$branch \prec^p tree$$

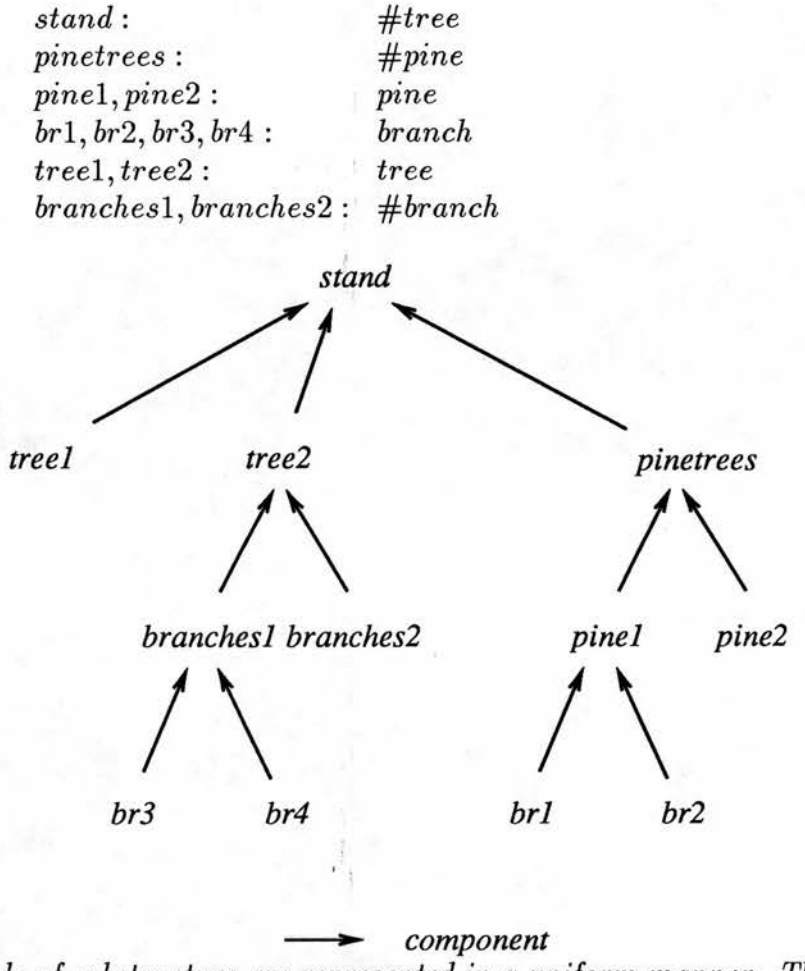
This induces the possible component relation, shown in figure 5–3. This fully characterises the space of possible component relations for this simple example. The root node of the possible component hierarchy is *#indiv*.

Suppose an ecologist is interested in modelling a forest stand with substructure. Suppose the stand consists of various individual trees as well as a substand



The possible component relation is wholly induced using the notion of a collection in conjunction with the sort and possible part hierarchies. It defines the space of possible components.

Figure 5–3: Induced Possible Component Relation



All three different kinds of substructure are represented in a uniform manner. The nature of a particular substructure relationship is inferrable form the types of the entities.

Figure 5–4: Substructure of a Forest Stand

consisting only of pines. Each tree may or may not have branches that are of interest. Different sets of branches might receive different amount of sunlight, and thus must be distinguished. Figure 5–4 shows such a component hierarchy for this hypothetical stand. Although there are different kinds of component relationships, all are components and may be created and viewed as such by the user without having to worry about the nature of the component relationship, nor the types of the components or wholes. The possible component relation helps ensure that only sensible substructure is defined. In § 5.6.6 we discuss the use of quantification and indexing concisely to represent substructure of a more uniform nature.

5.6.4 Homogeneity

In Elklogic, the notion of homogeneity is easily defined. A type T is homogeneous iff $T \subset^p T$. Note that normally, this is not the case, (*e.g.* $tree \not\subset^p tree$). All set types are homogeneous by rules 5.27,5.31, and 5.32, but all sorts are assumed to be non-homogeneous. As noted in § 2.5.4.6 some kinds of entities are homogeneous (*e.g.* a region). Thus we require $region \subset^p region$ which allows two entities of the same sort to be related by $\subset$. This may be accomplished by specifying $region \prec_\circ^p region$. This allows a region to be a component of a region just as $branch \prec_\circ^p tree$ allows a branch to be a component of a tree. Of course we still require that $tree \not\subset^p tree$; this is true so long as $tree \not\prec^p tree$. Summarising,

- For all sorts S, the type $\#S$ is homogeneous.
- A sort S is homogeneous if and only if $S \prec^p S$.

This *de facto* classifies homogeneous types into two categories: sets and sorts. Homogeneity is related to the continuous/discrete distinction. Most continuous entities appear to be homogeneous, but the reverse it not true. For instance, a *region* is mathematically continuous, but a finite set is not. However in ElkLogic, because there is no need to specify members of sets, sets behave much as if they were continuously decomposable.

In ELK, the only way to distinguish between a set which is apparently continuously decomposable, and other homogeneous entities like sand and sugar is by analysing the types of the possible component-whole relationships. Sets are necessarily viewed as consisting of (not necessarily explicit) components whose type is different than that of the set; the same is not true for sand. That is for any sort S, $S \subset^p \#S$; however there might be no sort S_c such that $S_c \neq sand$ and $S_c \subset^p sand$. It is possible for such an S_c to exist *e.g.* $S_c = grain$; this is accomplished by specifying $grain \prec_\circ^p sand$. Thus we can accommodate both the continuous and the discrete view of things like sand and sugar. This is also useful for *time*.

Time is the most important homogeneous concept that we are concerned with. It is continuously decomposable, yet we also have the need to view periods of time as of sets of *shorter* periods of time. For example, a $yr{:}year$ may be thought of simply as a year; alternatively it may be viewed as a set of 52 weeks, or a set of 365 sets of 24 hours. The representation for time is given in § 5.6.6.2.

All homogeneous types represent kinds of entities that (in the world) have the following property: the distinction between an entity of that kind, and of sets of entities of that kind is not fundamental, and can be blurred. For sets, this is captured by the fact that once flattened, all sets look the same (# is designed to capture this). For a non-homogeneous sort S, all the different subtypes of $\#S$ (*e.g.* $set(S)$) do not represent fundamentally different kinds of entities. For a homogeneous sort S, this property is reflected by the fact that a set of entities of type S may still be viewed as an entity of type S. Thus, for a homogeneous sort S, the types S, $\#S$, and $S^{\odot}$ do not represent fundamentally different kinds of entities.

We discuss homogeneity further in § 8.6.2.2 where we compare our theory with others designed specifically for representing the concept of continuity and being able smoothly to shift perspectives from discrete to continuous and vice versa.

5.6.5 Discussion

We now illustrate how the abstraction provided by # is useful when defining substructure. Suppose an instance of a lion population is created. Suppose that later this population is subdivided into two subpopulations. After the substructure is defined its type is no longer correct and must be updated. Formally:

Initially :	$ln\_pop{:}set(lion)$	
Subsequently :	$pride1{:}set(lion)$	$pride2{:}set(lion)$
	$pride1 \subset_{\circ} ln\_pop$	$pride2 \subset_{\circ} ln\_pop$
Now :	$ln\_pop{:}set(set(lion))$	

Suppose there was an individual lion (say $man\_eater{:}lion$) that was deemed significant enough to be included explicitly in the lion population (*i.e.* $man\_eater \subset_{\circ} ln\_pop$). Now, the type of ln_pop is $set(lion \sqcup set(lion))$. If the prides were further subdivided, the type would change again. This is unsatisfactory. From the users point of view, a population of lions is a still a population of lions however it may be subdivided. Thus, at some level of abstraction, its type should not change. Collections that we type using # give us exactly that level of abstraction. Because we only allow explicit creation of sets using # there is no need to update types of entities as their substructure changes.

The least types of collections are defined implicitly by the component relation. Though a simple procedure exists which can compute least types, (implemented in

Prolog), we have not found it necessary to use it. All we ever need to know about the more specific type of an entity of type $\#S$ is whether it can be expressed as $set(S)$, $set(\#S)$, $set(set(\#S))$, etc. This is dynamically determined by examining the substructure of the collection which is defined using $\subset$ (see example on page 140).

$\subset^p$ and $\prec^p$ are examples of a class of *permission* relations whose role is to ensure consistency in the description of the ecological system, not to describe it. This is the basis for distinguishing the general/ecological and the ecological system levels. This class was mentioned in chapter 4. Another example is an implicit relation called *entity_type* which characterises what all the entity types are. Formally,

$$\begin{aligned} entity\_type &: \mathcal{P}(entity) \mapsto bool \\ \forall T \sqsubseteq entity.\ &entity\_type(T) \\ \forall T \not\sqsubseteq entity.\ &\neg entity\_type(T) \end{aligned} \tag{5.33}$$

These relations are all used similarly. For example, if $\neg entity\_type(wazoola)$ then it is impossible to have instances of wazoolas in the description of the ecological system. Similarly, because $claw \not\subset^p tree$, we cannot say that some particular claw is a component of some particular tree.[5]

The analogy with sorts can be taken further. The sort *animal* is a subset of all entities in the world. Analogously, $paw \subset^p lion$ can be taken to be the subset of all paw-lion part-composite relationships in the world. In mathematical terms, just as *animal* is the subset of all instances of the type *entity*, $paw \subset^p lion$ is a subset of all the ordered pairs which define the $\subset$ relation.

We have certainly not solved the whole problem of representing substructure. One important flaw in the current definition of $\subset^p$ arises when there are subsorts for parts. For example, if $pine\_branch \sqsubset branch$ and $branch \prec^p tree$, and $maple \sqsubset tree$ then $pine\_branch \subset^p maple$ by application of *pos_member*, *d_pos_subdiv*, and *pos_coll_part* and transitivity. Specifically:

$$pine\_branch \subset^p_\circ \#pine\_branch \subset^p_\circ \#branch \subset^p_\circ tree$$

---

[5] In the implementation there is an explicit unary relation *sort* such that $\forall T \sqsubset indiv.sort(T)$. Thus $\mathcal{S}$, the set of all sorts is represented implicitly. *entity_type* is implicitly defined in terms of *sort* and $\#$. Types using *set* explicitly are not used except to check for computing things like average, maximum as discussed above.

This is clearly wrong. The flaw is in the transitive inference jumping across the second case. Thus the inference embodied in $\subset^p$ with respect to the sort and possible part hierarchies is not sound.

Because of how $\subset^p$ is used, this is not overly damaging. It is more important that it permits all sensible substructure than to prohibit all senseless ones. The presence of these flaws means that in certain cases, the system will not be able to spot inconsistencies. Although not optimal, this situation is not unusual. A similar situation exists with the qualitative reasoner QSIM which can give rise to spurious results (see [Kuipers, 1986]).

Even if $\subset^p$ was perfect (*i.e.* by allowing all and only sensible component-whole relationships with no computational redundancy) this is not sufficient. There are other things that must be guarded against when defining substructure. These include:

- *subdivisions:*
  - a set cannot be a component of a set which is a proper subset of it.
- *overlapping:*
  - two lions ought not have the same leg.
  - two trees ought not have the same branch. *but:*
  - two sets may share members
  - two geographic zones may overlap and have shared regions
  - two one week intervals can share a day if they are not contiguous.
- *how many parts:*
  - a person should not have more than one head
  - a lion should have no more than 4 legs

Regarding subdivisions, $\subset^p$ only checks that the type of the sets are compatible, it does not know about actual members. Suppose $ln1, ln2 \subset pride1$, and $ln1, ln2, ln3 \subset pride2$, where $ln1, ln2 : lion$, and $pride1, pride2 : \#lion$. Even though $pride2 \subset pride1$ is otherwise permitted because $\#lion \subset^p_\circ \#lion$, it should be forbidden.

In general overlapping appears to make sense for sets and for homogeneous individuals and should be allowed. By default, all sorts are assumed to be non-homogeneous and thus no overlapping should occur. Where overlapping should be permitted, this may be accomplished using the possible part relation as noted previously. If there is overlapping of wholes, $\subset$ is a graph, otherwise it is a tree.

The current version of ELK does not check for overlapping. However, § 5.6.4 and this discussion indicate that this feature could be added with minimal conceptual effort.

As part of the general/ecological knowledge base, we should further say that a person has exactly 1 head, and a lion exactly 4 legs[6]. However, in the conceptual model of a particular ecological system, we certainly do not want to insist on the explicit creation of components corresponding to all parts of complex entities. This is analogous to not insisting that a set have explicit members. As we can create a pride with no explicit individual lion members so also can we create a lion with no individual legs. We discuss this matter further in § 8.6.2.1.

5.6.6 Indexing

The above mechanisms are very general and suitable for representing a wide variety of substructure relationships. However, there are various special cases which we still need to cater for. For example, there might be 20 similar trees in a stand. We might wish to subdivide a pride by sex, colour, and age. Having manually to create 20 tree entities ($tr1, tr2, \ldots$) is neither convenient nor concise. Similarly, we would not wish to require specifying all the different subdivisions manually. In this section we describe various indexing techniques to make this more convenient. All that is required is to specify the number of trees to be created, or what the different values of sex, colour, and age groups are.

5.6.6.1 Pure Indexing

We use the notation I_i to refer to the set of integers from 1 to i. To create 20 instances of *tree*, each a component of $stand{:}set(tree)$, we do the following:

$$\begin{aligned} tr: \quad & natural \mapsto tree \\ \forall I \in I_{20}. \quad & tr(I){:}tree \\ \forall I \in I_{20}. \quad & tr(I) \subset_{\circ} stand \end{aligned}$$

In § 7.4.1 we show how this specification may be created using ELK. Other things are possible, however we do not have interface commands for all of them. For

[6] We are not concerned with rare cases of two-headed people or 3-legged lions.

example, using this technique, in conjunction with $\subset^p$ it is easy to specify that all the trees in some set have 10 branches. Also, different trees may have a different numbers of branches. For example:

$$\begin{array}{ll}
\forall I \in I_{19}.\forall J \in I_{10} & brnch(I,J){:}branch \\
\forall I \in I_{19}.\forall J \in I_{10}. & brnch(I,J) \subset_o tr(I) \\
\forall J \in I_5 & brnch(20,J){:}branch \\
\forall J \in I_5 & brnch(20,J) \subset_o tr(20)
\end{array}$$

This is a purely syntactic technique; there is no semantic information contained in the term denoting the entity. The I'th tree is not inherently different from the J'th one, although in the real world and/or in the simulation they may have different growth rates, size, etc. Another example where indexing may effectively be used is for representing time substructure. The representation for time is discussed next.

5.6.6.2 Representing Time

Suppose we wish to run a model for a period of 7 years, where each year is subdivided into 12 months. We can represent this as follows. First we introduce basic time sorts for each time unit (*e.g.* *year*, *month*, *week*). These are subsorts of *time*. Let $7yrs{:}\#year$ represent the overall time period. It has 7 component years $(yr(1),\ldots,yr(7))$, each of which has 12 component months. We first describe the obvious way to represent this which was used in [Bundy & Uschold, 1989]. After that, we note some problems with this approach and adapt it accordingly. Let $mnth(I,J)$ denote the J'th month of the I'th year. Formally:

$$\begin{array}{ll}
yr: & natural \mapsto year \\
mnth: & natural \times natural \mapsto month \\
\forall I \in I_7. & yr(I){:}year \\
\forall I \in I_7.\forall J \in I_{12}. & mnth(I,J){:}month \\
\forall I \in I_7. & yr(I) \subset_o 7yrs \\
\forall I \in I_7.\forall J \in I_{12}. & mnth(I,J) \subset_o yr(I)
\end{array}$$

Depending on the needs of the modeller, each month might further be subdivided (*e.g.* into days). To permit this substructure, we require that $day \subset^p month$, $month \subset^p year$, etc. Because $7yrs{:}set(set(month))$, we also require $\#month \subset^p year$. We accomplish this by using the $\prec^p$ hierarchy. Just as $branch \prec^p tree$ implies that $\#branch \subset^p tree$ so also does $month \prec^p year$ imply $\#month \subset^p year$,

as required. We do not want to allow a year to be part of a day and thus must prohibit larger time units from being components of smaller ones. To achieve the desired effect, we place all the time sorts *in order* in the part hierarchy. Formally:

$$day, week, month, year \sqsubset^s time$$
$$day \prec^p_\circ week \prec^p_\circ month \prec^p_\circ year$$

We require that these times are totally ordered. To achieve this, we define a unary function *next* from times to times. This behaves as a successor function; the order is implied. Formally:

$$next : time \mapsto time$$

$$next(mnth(I, J)) = \begin{cases} mnth(1, J+1) & \text{if } I = 12 \\ mnth(I+1, J) & \text{otherwise} \end{cases}$$

Although we have stated that the term $mnth(3, 2)$ denotes the 2nd month in the 3rd year, the term does not itself carry enough information to produce this semantic interpretation. Furthermore, not all the substructure is explicit in the terms. In particular, the representation of the various parts of the substructure of the overall interval are distinct ($yr(I)$ is not obviously associated with $mnth(I, J)$). We introduce an alternate scheme to address these shortcomings.

The expression, $tim\_dim\_fn_2(7yrs, year, month)(3, 2)$ denotes the 2nd month of the 3rd year. The third year is denoted by $tim\_dim\_fn_1(7yrs, year)(3)$. Not only can the meaning be easily derived from this representation, but so also may the substructure be inferred. The former is a component of the latter by virtue of sharing common values for the first index. In general, a substructure relationship can be inferred when the first j indices are identical, where $j \leq n$. This inference is analogous to inferring that $\{Ln{:}lion | sex = male, colour = brown\}$ is a subtype of $\{Ln{:}lion | sex = male\}$ in many logics with subtypes [Cardelli, 1989] (*i.e.* purely from the structure of the expressions). Formally:

$\forall N, N_1, N_2, \ldots, N_n : natural. \forall T, T_1, T_2, \ldots T_n \sqsubset time. \forall E{:}T$
where $T_n \prec^p T_{n-1} \prec^p \ldots \prec^p T_1$
$tim\_dim\_fn_n : T \times \mathcal{P}(T_1) \times \ldots \times \mathcal{P}(T_n) \mapsto (natural^n \mapsto T_n)$
$tim\_dim\_fn_j(E, T_1, T_2, \ldots, T_j)(N, N_1, N_2, \ldots, N_j) \subset$
$\quad tim\_dim\_fn_n(E, T_1, T_2, \ldots, T_j, \ldots, T_n)(N, N_1, N_2, \ldots, N_j, \ldots, N_n)$

We have made a formal distinction between so-called primitive time entities ($yr80{:}year$) and sets of times, ($80s{:}set(year)$). Although this is useful for enabling us to perform certain kinds of computation (*e.g.* an average), it does *not* reflect a real difference between a year and a set of years in the same way that a lion is different from a set of lions. Both $yr(1)$ and $7yrs$ can be viewed as specific periods of time of some length. Unlike for $ln{:}lion$ and $pride{:}set(lion)$, there is nothing in principle that we can do with one but not the other.

To compute the average weight of a single lion (at a some time) is meaningless. Average only applies to sets. However, it *does* make sense to compute the average weight (of an entity) in $yr(1)$ because we can further subdivide it into entities of the same type (*i.e.* *time*). We can subdivide any year into months so that it is entirely reasonable to say that $yr(1) : set(month)$, or $yr(1) : set(set((day))$ if the year were divided into months each of which were subdivided into days. We cannot further subdivide *lion* into entities of the same type. Because time is homogeneous, *time*, $\#time$, and $time^{\odot}$ are not fundamentally different (see end of § 5.6.4).

Consider the following expressions:

$$average(\lambda E{:}phys\_obj.weight(E,T),SetE) \tag{5.34}$$

$$average(\lambda T'{:}time.weight(E,T'),T) \tag{5.35}$$

$$maximum(\lambda T_1{:}time.average(\lambda T_2{:}time.weight(E,T_2),T_1),T) \tag{5.36}$$

5.34 is well typed only if $SetE{:}set(phys\_obj)$. It may be computed only if $SetE$ has explicit members. In either case, it means the same thing. Because time is homogeneous, 5.35 should be well typed whether or not T is explicitly of type $set(time)$. If $T{:}year$ in expression 5.35, the expression denotes the average weight of entity E for the one year period represented by T. If T has no substructure, the average may not be computed in the usual way. If T has a number of instances of *month* as components, then the average may be computed, but its meaning is the same. It would not be possible to compute expression 5.36 unless additionally each month was further subdivided (say into days). This is one case where the system must know more about the type than can be represented using the # notation alone. It does not suffice to know that $7yrs{:}\#year$, in order to calculate explicitly an average, or the maximum of several averages, the system must know whether $7yrs$ is an instance of $set(time)$ or $set(set(time))$ respectively. If $7yrs$

is a set of years, each subdivided into months, then $7yrs : set(year)$ and $7yrs : set(set(month))$.

5.6.6.3 Attribute-Based Substructure

By attribute-based substructure, we mean we subdividing a set of entities based on the values that these entities have for certain attributes (*e.g.* to subdivide a population in to two according to sex). The usual way to represent this substructure is to define subtypes corresponding to different values of different attributes. For instance, the type of male lions might be defined by: $\{L{:}lion | sex(L) = male\}$ [Cardelli, 1989]. This approach requires sophisticated browsing techniques to deal with the following:

- it gives rise to an infinite lattice, when all values of all attributes are incorporated.
- many of the new types will no longer correspond to natural kinds in any useful sense.
- most of the subtypes will be used infrequently if at all

We have chosen a different approach which requires users only to browse through the sort hierarchy which will usually consist of natural kinds. There is no need for even simple induced types like $set(lion)$ to appear explicitly. Instead of representing male lions as a type, we create instances of *lion* and incorporate the substructure in the name of the instance (*e.g.* $ln(male){:}lion$)[7]. It is rare that we should be interested in modelling a single male lion, but we may very well choose to subdivide a pride into male and female sub-prides. We might represent this by $pride(male) : \#lion$ and $pride(female) : \#lion$. Rather than being a constant of type $\#lion$, *pride* would be a function mapping values of the sex attribute to entities of type $\#lion$.

Before we give the formalities, we describe a mechanism for typing values for attributes like colour and sex whose value spaces are specified as finite sets. There are two possibilities. First, if the values are of a commonly used sort, it may be convenient to give the sort a name and place it in the sort hierarchy under

---

[7] The idea of incorporating substructure information in the instance name rather than in the type system was Alan Bundy's.

value. This is often convenient, but not always. Consider the attribute *colour*. In English, the word 'colour' is ambiguous in that it refers both to the attribute (*e.g.* colour of a lion) and the value of the attribute (*e.g.* brown is a colour). One might use 'colours' for the value sort and 'colour' for the attribute, however this is not a satisfactory solution in general. It demands that two different primitives are used to refer to different aspects of the same underlying concept. This goes against our general philosophy of retaining connections. As a practical matter, it may not always be easy to think of sensible names, and is a nuisance to have to do so.

An alternate approach is to automate the typing of the values for such attributes. We use the sort function v which given the name of an attribute, returns a subsort of *value*. Formally:

$$\forall V \sqsubset value.\ v : (ecol\_ent \times time \mapsto V) \ \mapsto \ V$$

It is only required when the value space is finite, and not itself a sort. By contrast, the value space for weight is the set of positive reals denoted by the sort: *positive*. Using facilities for defining attributes to be described in chapter 6, the value space for the attribute *colour* may be given as $\{green, brown\}$. This implicitly specifies the following:

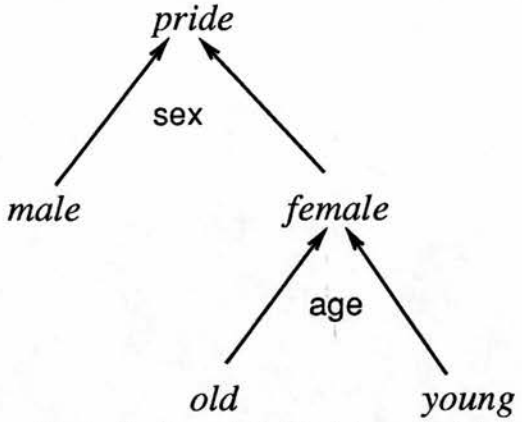
$$v(colour) \sqsubset value$$
$$green{:}v(colour);\ brown{:}v(colour)$$

We now give the type of *pride* in the example:

$$pride : v(sex) \mapsto \#lion$$

There is a potential practical problem of redundancy if more than one attribute uses the same sort of values (*e.g.* *eye_colour*, *skin_colour*). We discuss how this may be dealt with when we give examples in chapter 6.

Suppose we wish further to subdivide the female pride (but not male) into two age groups. In general, specifications for complex nested substructure may be given in an AND/OR tree. Let the age groups be *adult* and *cub*. This specification gives rise to, three entities, $pride(female'(adult))$, $pride(female'(cub))$, and $pride(male)$. Here we have lost the formal connection between $female$ and $female'$. To recapture this we introduce a series of higher-order functions analogous to $tim\_dim\_fn_i$; these are $att\_dim\_fn_i$ (i indicates the number of arguments



This specifies that the pride is to be subdivided by sex into male and female subdivisions. The female, but not male subdivision is further subdivided by age into two groups: old and young.

Figure 5–5: Specifying Attribute-Based Substructure

of the output function). It automates the creation and typing of functions from subdimensions to dimensions given an AND/OR tree which specifies the substructure (see figure 5–5). We use $att\_dim\_fn_1$ to define $female'$ automatically in terms of $female$. For example $pride(att\_dim\_fn_1(female, age)(cub))$ might denote the collection of female cubs, where *cub* is defined as some interval, (*e.g.* $[0,1] : set(positive)$). We thus retain an explicit connection with $female$, and avoid proliferation of names like $female'$ which would get out of hand except in very simple cases for nested substructure.

There is one last minor point. If a user initially defines *pride* as a set of lions, and later decides to define substructure, the type of *pride* would have to change. *pride* would no longer denote the pride. This creates a problem for the user interface which can be resolved in one of two ways. First, the user may be prevented from referring to *pride* as they initially did. Second, they can still do so, but the system will have to keep track of what they really mean. The first is unnecessarily restrictive to a user and thus unacceptable. The second forces the system to create another instance which refers to the whole pride and must always be careful to convert *pride* to this instance whenever entered by the user. This is workable, but we have chosen an alternative which has neither of these problems.

pride stays exactly as it was, and we wrap $att\_dim\_fn_1$ around it at the outer level. Thus, we represent the collection of female cubs as:

$att\_dim\_fn_1(pride, sex)(att\_dim\_fn_1(female, age)(cub))$. If both males and

females were subdivided by age, the female cubs would be represented by:
$att\_dim\_fn_2(pride, sex, age)(female, cub)$. Formally,

$$
\begin{aligned}
&\forall T, T_1, T_2, \ldots, T_n{:}ecol\_ent. \\
&\quad \forall V_1, V_2, \ldots, V_n{:}value. \\
&\qquad att\_dim\_fn_n \;:\; (T \times (T_1 \times time \mapsto V_1) \\
&\qquad\qquad \times (T_2 \times time \mapsto V_2) \\
&\qquad\qquad \times \ldots \times (T_n \times time \mapsto Vn)) \mapsto \\
&\qquad\qquad (V_1^{\odot} \times V_2^{\odot} \times \ldots \times V_n^{\odot}) \mapsto T \\
&\qquad att\_dim\_fn_1(pride, sex) \;:\; v(sex) \mapsto \#lion \\
&\qquad att\_dim\_fn_2(female, age, colour) \;:\; positive^{\odot} \times v(colour) \mapsto v(sex) \\
&\qquad female' = att\_dim\_fn_1(female, age) \;:\; positive^{\odot} \mapsto v(sex)
\end{aligned}
$$

So, except at the outer level, $att\_dim\_fn_i$ is used to map values of subdimensions to dimensions. At the outer level, it maps values in the first dimension to the type of the subdivided entity itself. This can be thought of as the value of the degenerate 0th dimension of substructure). The types of the n-ary functions produced by $att\_dim\_fn_i$ (*i.e.* using the $\odot$ notation) imply that there are two kinds of indices to these functions. One is individual values (*e.g.* $male{:}value$); the other is sets of values (*e.g.* $cub = \{0, 1\}{:}\#value$).

The complete specification of the substructure in the above example is represented below:

$$
\begin{aligned}
&\forall Sex \in \{male, female\}.att\_dim\_fn_1(pride, sex)(Sex) \subset_{\circ} pride \\
&\forall Age \in \{adult, cub\}.att\_dim\_fn_1(pride, sex)(att\_dim\_fn_1(female, age)(Age)) \\
&\qquad\qquad \subset_{\circ} att\_dim\_fn_1(pride, sex)(female)
\end{aligned}
$$

The user is of course never expected to create, examine, or modify this directly. The natural way to view this substructure is as an and/or tree (see figure 5–5). In § 7.4.2 we describe the interface by which a user creates and modifies this and/or tree. ELK uses the specification of that tree, to implicitly specify the terms given above. Those terms directly reflects the meaning of the instances and the overall substructure of the pride. For example:

$\forall T_m{:}time.$

$\forall X \in att\_dim\_fn(pride, sex)(male).sex(X) = male$

$\forall X \in att\_dim\_fn(pride, sex)(att\_dim\_fn(female, age)(cub)).$

$sex(X) = female \wedge age(X, T_m) \in cub$

This does not mean that a cub will never grow up, but that when does, it is no longer a cub. In general:

$\forall T_m{:}time.$

$\forall T \sqsubset \#ecol\_ent.\forall V_1, V_2 \sqsubset value.$

$\forall Val_1{:}V_1.\forall Val_2{:}V_2.\forall Vals : \#V_1$

$\forall A_1{:}ecol\_ent \times time \mapsto V_1.\forall A_2{:}ecol\_ent \times time \mapsto V_2.$

$\forall E_1, E_2, E_3{:}T.$

$\forall att\_dim\_fn_1(E_1, A_1)(Val_1){:}T.$

$\forall att\_dim\_fn_1(E_2, A_1)(Vals){:}T.$

$\forall att\_dim\_fn_1(E_3, A_1)(att\_dim\_fn_1(A_2, Val1)(Val_2)){:}T.$

$\forall X \in E_1.A_1(X, T_m) = Val_1$

$\forall Y \in E_2.A_1(Y, T_m) \in Vals$

$\forall Z \in E_3.A_1(Z, T_m) = Val_1 \wedge A_2(Z, T_m) = Val_2$

and

$\forall V, V_1, V_2 \sqsubset value.$

$\forall Val{:}V.\forall Val_1{:}V_1.\forall Val_2{:}V_2.$

$\forall E_1, E_2{:}\#ecol\_ent.$

$\forall A{:}ecol\_ent \times time \mapsto V.$

$\forall A_1{:}ecol\_ent \times time \mapsto V_1.$

$\forall A_2{:}ecol\_ent \times time \mapsto V_2.$

$att\_dim\_fn(E, A)(Val) \subset_\circ E$

$att\_dim\_fn(E, A_1)(att\_dim\_fn(Val_1, A_2)(Val_2)) \subset_\circ att\_dim\_fn(E, A_1)(Val_1)$

This may continue in a recursive manner to allow for arbitrarily deep nesting.

Possible Dimensions

Usually, the entity being subdivided will be a set. The possible dimensions for defining substructure include the attributes of the base sort (S), plus those of the corresponding collection type ($\#S$). Thus a lion population may be subdivided according to *age* which applies to *lion*, or $qnam\_ent(average, age)$ which applies

to $set(lion)$. If age is used, then $att\_dim\_fn(ln\_pop, age)(cub)$ denotes the subset of the lion population which are cubs. If $cub = [0,1]$, then

$$\forall T_m{:}time.\forall L \in att\_dim\_fn(ln\_pop, age)(cub).\ age(L, T_m) \in [0,1]$$

If average age is used as a dimension for defining substructure, then the population being subdivided must be a set of sets. The sets in this set of sets are of [at least] two kinds. One kind is a set characterised by the average age of the member lions being between 0 and 1. The set of all sets of this kind is denoted by:

$$att\_dim\_fn_1(ln\_pop, qnam\_ent(average, age))(cub)$$

Furthermore,

$$\forall T{:}time.\forall Lp{:}\#lion.$$
$$Lp \subset att\_dim\_fn(ln\_pop, qnam\_ent(average, age))(cub)$$
$$\rightarrow\ qnam\_ent(average, age)(Lp, T) \in [0,1]$$

Note also that although we require that *ln_pop* is a set of sets in the latter example, it need not have explicit member sets. This is a consequence of the fact that we never require explicit members for any collection. The only possible exception would be if an explicit calculation were to be performed for maximum, average, etc. This is up to the user to decide.

If a set of sets of sets of lions is being subdivided, then the attribute $qnam\_ent(maximum, qnam\_ent(average, age))$ could also be used as a dimension for defining substructure. This would enable us to distinguish different sets of sets of lions on the basis of what the maximum average of the member sets of each set of sets was. It is unlikely that this level of nesting would ever be required, but due to the uniformity and generality of the representation for attributes, it is allowed. ELK is able to unpack the expressions and determine the meaning of the logical terms denoting the component entities.

This representation is very similar to that used by [Mellish, 1988]. Both are notational variants of property lists. Our representation shares the advantage of Mellish's in that the syntax of the terms directly reflects the substructure represented. However, in the latter, the terms are purely syntactic. It is not possible to give them a semantics and logical typing without introducing redundancy as described above. Using the $att\_dim\_fn_i$ representation, the meaning of the terms is immediately available. Furthermore, it is trivial to infer the substructure relationships between the entities that they denote.

Attribute-based substructure is only concerned with member and subset, not part-composite. However we have incorporated all kinds of substructure, attribute-based and otherwise, into a single framework using the $\subset_o$ relation and its transitive version $\subset$. The attribute-based method which uses indexing is only useful when there is considerable uniformity in the substructure being defined; it uses $\subset_o$ indirectly. For defining arbitrary substructure relationships the indexing technique is inappropriate; $\subset_o$ is used directly.

5.7 Representing Ecological Model Variables

Finally, we consider how to represent model variables in such a way as to be able to recover ecological information to explain the model. The obvious way to do this is to let the functions corresponding to the attributes also be the model variables. We shall refer to this as *attribute-variable conflation*; it is the approach taken in [Bundy & Uschold, 1989]. Thus $number(wb\_pop, T)$ would be the model variable corresponding to the number of wildebeest.[8] Similarly, $weight(grass, T)$ would be the model variable for the weight of grass. This approach goes directly against one of our main 'techniques', namely the separation of the conceptual model (*i.e.* ecological system description) and the simulation model into distinct representations. When first introducing model variables in § 2.4.1 we said that variables are idealisations of attributes not identical to them. We have argued extensively in chapter 4 why this distinction is important. We review the main points briefly here and elaborate on some more specific points. This distinction facilitates meeting the following requirements:

1. to facilitate model comprehension
2. to identify the idealisation search space
3. to ensure consistency
4. to retain explicit connections between frequently used concepts and thus avoid a proliferation of primitives
5. to allow proper typing of parameters

---

[8] Strictly speaking, the variable is a unary function and should be represented as: $\lambda T{:}time.number(wb\_pop, T)$

6. to have a direct and obvious representation of model variables

The first three points require no additional comment here. On the fourth point, suppose we wish to model the attribute *weight* in different ways as described in § 2.5.5.5. There would no longer be a single function *weight*, but possibly many, each with different (albeit related) value spaces. Formally we might have:

$$\begin{aligned} weight &: phys\_obj \times time \mapsto positive \\ weight' &: phys\_obj \times time \mapsto \{small, large\} \\ weight'' &: phys\_obj \times time \mapsto \{small, medium, large\} \end{aligned}$$

Information about the general concept of weight must be duplicated three times with slight variations. We no longer have a single fixed concept of weight (*i.e.* the real world attribute) from which the idealised versions may derive. This makes it harder to use general purpose machinery to give an ecological account of the model variables, harder to identify the idealisation search space, and harder to ensure consistency.

Points 5 and 6 are less important, but worth mentioning. Many attributes which vary in real life may be idealised as being constant (*i.e.* modelled as parameters). In our example simulation model, the number of predators is fixed. In table 2–1 the model variable for this was correctly typed: $n\_pred : real$, but contained no ecological information. If we conflate attributes and variables, we would represent this model variable as:

$$\lambda T{:}time.number(predators, T) \;:\; time \mapsto real$$

This gives us the ecological information that we require, but it is the wrong type. To assign such model parameters values requires quantification over all times. This is undesirable.

Finally, model variables are of chief importance in the model; the representation should capture them in a natural and obvious way. All proper model variables are unary functions from time to some value space. Because most attributes are binary functions mapping entities and time to some value space, it is not immediately obvious what the actual model variables are. For example the expression: $number(Set, T)$ may be used in the context of many different model variables and parameters. To get an explicit representation of a single model variable, we must instantiate one argument and use λ-abstraction as we did above. This is a minor

annoyance. This only affects the system developers and is largely irrelevant to the users who would not normally see the complex logical terms and expressions.

We avoid all of these problems and meet our requirements by using higher-order functions to define an explicit mapping from ecological attributes [and process effects] to model variables and/or parameters. We have three of these:

- $attvar\_fn$ maps ecological attribute and ecological entity pairs to proper model variables

  e.g. $attvar\_fn(number, wb\_pop) : time \mapsto positive$ is the model variable corresponding to the number of wildebeest.
- $attparm\_fn$ maps ecological attribute and ecological entity pairs to parameters

  e.g. $attparm\_fn(number, predators) : natural$ is the model parameter corresponding to the number of predators.
- $effvar\_fn$ maps ecological effects to effect variables

  e.g. $effvar\_fn(wb\_eaten') : time \mapsto positive$ is the partial rate variable corresponding to the number of wildebeest eaten by the predators per year. Recall that an effect variable may be a partial rate variable, an intermediate variable, or a parameter depending on how the process is idealised.

In § 6.4.2.3 we say how processes are defined; there we give a proper representation of the real effect wb_eaten' in terms of processes, attributes, and entities. The details of the types of the variables represented above would vary according to the idealisation choices made. The types of these functions are given below. We give the abstract types also, to elucidate the relationship between ecological and modelling concepts not captured by the object-level types. In each case, V is the idealised version of the value space V'.

$$\forall E{:}T.T \sqsubset ecol\_ent.V, V' \sqsubset \#value$$

$$attvar\_fn : attribute \times entity \mapsto propvar$$

$$attvar\_fn : ((T \times time) \mapsto V') \times E \mapsto (time \mapsto V) \tag{5.37}$$

$$attparm\_fn : attribute \times entity \mapsto parameter$$

$$attparm\_fn : ((T \times time) \mapsto V') \times E \mapsto V \tag{5.38}$$

$$effvar\_fn : effect \mapsto propvar$$

$$effvar\_fn : (time \mapsto V') \mapsto (time \mapsto V) \tag{5.39}$$

Usually, $V = V'$ or is a set of equivalence classes in V'. For example, each of *small, medium* and *large* are equivalence classes for *real*.

We meet requirement 1 (from list on page 172) because these expressions are directly expressed in ecological terms; it is a trivial syntactic matter to generate English text explaining the ecological meaning of each model variable. ELK does exactly this (see § 6.8 for examples). Additionally, it is possible to explain where and to what extent idealisation has occurred (see discussion on meeting requirement 4 below).

We meet requirements 2 and 3 by maintaining explicit separation of the ecological level and the runnable-model level. There are various idealisation decisions that may be made going from the conceptual model to the simulation model. One of these is the choice of value space; another is how to model the effect of a process. We ensure that the choices made are consistent with the ecological system. For example, we would not allow predation to cause the number of the prey population to increase.

The separation of the ecological system description and simulation model also facilitates meeting requirement 4. To illustrate this, the three model variables corresponding to the three idealisations of the weight attribute might be:

$$\begin{aligned} attvar\_fn(weight, phys\_obj1) : time &\mapsto positive \\ attvar\_fn(weight, phys\_obj2) : time &\mapsto \{small, large\} \\ attvar\_fn(weight, phys\_obj3) : time &\mapsto \{small, medium, large\} \end{aligned}$$

We create a distinct model variable for each entity which has a certain attribute. The decision of which value space to use is made independently for each. The default values are taken from the general/ecological knowledge base where the attribute weight is characterised. We use meta-level constructs for this; the details are in chapter 6. Thus we record in a single place the knowledge about an attribute; this knowledge may be used again and again. This avoids proliferation of primitives (like *weight'*, *weight''*), facilitates both consistency checking, and the use of general mechanisms for automatic model documentation.

We meet requirement 5 because parameters are represented as constants, not functions on time.

We meet requirement 6 because each ecological model variable is associated with exactly one '*_fn*' expression which has the right type.

The upshot of this is that rather than substantially change the representation of the runnable model, we can *augment* it. The runnable model that we create using ELK may be exactly as in figure 2–4. The types may be exactly as in table 2–1. What differs is that we record a wide variety of extra information which resolves the above problems in particular, and meets the many design requirements discussed in chapter 4. The nature of this extra information in the context of this simple model was outlined in § 2.5.1.

Although one is ecological information, and the other runnable-model information, *both attributes and model variables are object-level functions.* The distinction we require between ecological and modelling information is made at the meta-level. One class of meta-level constructs is used to create attributes, and another to create model variables (details in chapter 6).

5.7.1 Induced Model Variables

The function qualifiers (*e.g. average*) induce new object-level functions be they attributes or variables. Thus, even with no ecological information the system can provide a limited amount of support in generating new variables. Given the function n_wb, a user can select the induced model variable $qnam(maximum, n\_wb)$. This would be useful if there was a specific time period over which computing the maximum value of n_wb was interesting. They need only indicate which time periods they wish to have the computation performed with respect to. The system can do the rest. Formally:

$$
\begin{array}{lll}
n\_wb: & time & \mapsto positive \\
qnam(maximum, n\_wb): & set(time) & \mapsto positive \\
qnam(maximum, n\_wb)(80s) & = & max(\{n\_wb(yr80), \ldots, n\_wb(yr89)\})
\end{array}
$$

Where max is a unary function returning the maximum of a set of values. This support is much more worthwhile if maximum or other qualifiers are relevant for other variables and/or if the same time periods are useful for other purposes. If it was a one-off case, there would be little to argue in favour of using *maximum* instead of max when defining the equations. This illustrates the general principle that frequency of use determines the worthwhileness of additional expressive power.

If the user wishes to compute the maximum size of a few lion prides, the system can offer no support unless the lion prides are represented. The user will otherwise

manually have to specify what they require using max. Formally:

$$max\_no(yr89) \;=\; max(\{size\_pride1(yr89), size\_pride2(yr89), \ldots\})$$

Whether we represent entities or not, we need separate variable names for the size of each pride. Where we win by having entities is that the system can do some of the work. A user would merely have to tell the system to create a variable corresponding to the maximum *number* of a particular set of prides at some time. The end result is the same. Formally:

$$pride1, pride2, pride{:}\#lion; \quad pride1, pride2 \subset pride; \quad pride = \{pride1, pride2\}$$
$$attvar\_fn(qnam\_ent(maximum, number), pride) : set(\#lion) \mapsto positive$$
$$attvar\_fn(qnam\_ent(maximum, number), pride)(yr89)$$
$$= maximum(\lambda P{:}\#lion.attvar\_fn(number, P)(yr89), pride)$$
$$= max(\{attvar\_fn(number, pride1)(yr89), attvar\_fn(number, pride2)(yr89)\})$$
$$= max(\{size\_pride1(yr89), size\_pride2(yr89)\})$$

Note that this implies that model variables must be defined corresponding to the *number* attribute of each entity. This could be done automatically by the system, or may have already been done by the user. We discuss the implementation issues in more detail in chapters 6 and 7.

It is important to note that induced attributes and variables are exactly analogous to regular ones in that they are treated in the same way for most purposes. For example, we might have a model variable corresponding to the induced attribute $qnam\_ent(average, fecundity)$ representing the average fecundity of some population. This is treated just like any other attribute of a set entity. The set may or may not have members. If not, then no computation of average can take place. A value assigned to a variable corresponding to this attribute represents the fecundity of an average individual in that population even though none of the members are explicitly represented. Fecundity is likely to be constant and thus represented as a parameter.

- Entity types
  - sorts: *real*, *lion*
  - simple set types: $set(lion)$, $set(set(integer))$
  - collection (arbitrary set) types: $\#lion$
  - arbitrary entity types: $indiv^{\odot}$
  - union types: $lion \sqcup set(lion)$
  - complement types: $indiv^{\odot} \setminus_t indiv$
- Relations:
  - *type relations*: relations that hold between types
    e.g. $\sqsubset$, $\subset^p$, $\prec^p$
  - *regular relations:* relations that hold between entities
    e.g. $\subset$, $\in$, *predation*
- First order functions:
  - *type function:* a function that returns a type
    e.g. *set*, $\#$, $^{\odot}$, $\sqcup$, $\setminus_t$.
  - *regular function:* a function that returns an entity
    e.g. *n_wb*, *weight*, *number*
- Higher order functions:
  e.g. *rate*, *maximum*, *average*, *qnam_ent*, *qnam_tim*

Figure 5–6: Type Classification

5.8 Summary and Conclusion

We have presented almost everything we need to know about the underlying formalism for the object-level language. Note that, the object/meta distinction is not what distinguishes ecological information from the runnable model. The object language is used to represent both. That distinction is made at the meta-level. Some of the constructs we have introduced are useful for both levels, some only for one or the other. The gross structure of the type hierarchy is given in figure 5–2. This is summarised with examples in figure 5–6.

With the exception of the regular functions and relations (defined in the figure), all of this comprises the immutable kernel of the object language. It includes mostly a framework with virtually no ecological vocabulary. In order to describe ecological systems and models, users modify the vocabulary by editing the sort hierarchy, creating and destroying instances, and creating and destroying functions and relations which correspond to attributes, model variables and processes. For

example, creating the model variables like *n_wb*, or attributes like *weight* results in new object-level functions. Thus, *the users play a significant role in defining the object language.*

We distinguish between *creating* and *defining* functions and relations. The former means to give it a type and add it to the vocabulary of the language. The latter means to supply the n-tuples. In the case of a function, this means saying how it is computed. The relation $\subset_o$ is created by the system, but defined by users. $\sqsubset^s_o$ is created and partially defined by the system and partly defined by the user. $\sqsubset^s$ is wholly and implicitly defined by the system in terms of $\sqsubset^s_o$. We allow for dynamic creation of new sorts, regular relations, and regular functions. The latter are almost always either model variables or attributes. We do not allow creation of any type functions, type relations, or new higher-order functions.

The most important object-level constructs that are explicitly created by users are regular functions corresponding to attributes and model variables. In the kernel, there are no model variables and only one attribute (*number*). We require meta-level constructs (alias Prolog facts and predicates) for creating and characterising object-level functions and relations. For example, we have an explicit meta-level construct (*att_def*) which is used to define attributes. We would use it to define *number* and *weight* (from § 5.4) as follows:

$$att\_def'(weight, phys\_obj, positive)$$
$$att\_def'(number, \#indiv, natural)$$

A prime (′) denotes that a simplified version of the actual ElkLogic construct is being used. Full details are in chapter 6 and summarised in appendix C.

It is important to stress that the underlying complexities of the theory presented in this chapter are not seen by end users. For example, the rather ugly $att\_dim\_fn_i$ constructs used to represent substructure are automatically created as the result of the user specifying an and/or tree. We provide a friendly interface whereby users can gradually specify complex nested expressions using *average*, *maximum* etc without ever seeing a parenthesis, or a λ. These and many other examples of how ELK is used are described in chapter 7.

It is important also to understand the advantages bought by the added complexity. The chief use of higher-order functions is to keep the number of primitives small. This allows general purpose facilities to generate textual documentation of the specification. The $\_fn$ higher-order functions make explicit the connection

between the ecological attributes and the model variables; this is one important way to ensure model comprehension. Another use of the higher-order functions, in conjunction with the type structure is to ensure consistency and reduce the modelling search space. These points have been discussed in chapter 4 and are further illustrated in the subsequent chapters.

This concludes the presentation of the underlying theory of ELK. In the next two chapters, we describe the implementation. There are two main aspects: representation and elicitation. Chapter 6 deals with the representation; in it we present a variety of meta-level constructs similar to att_def. We show how these constructs in conjunction with the ones described in this chapter may be used to describe the required information for our example model. Chapter 7 describes how ELK is used to elicit descriptions of ecological knowledge, systems, and simulation models from the user, and how to run the latter.

Chapter 6

ELK: Representation

6.1 Introduction

In designing and implementing ELK, we reformulate the ecological modelling formalisation problem. The main issues that we face in this context are:

- model comprehension
- expressive power
- conceptual distance
- assistance during elicitation

Model comprehension arose as a major requirement in the simulation domain. It is useful both:

- after a model is constructed, so that others may more easily understand the model and use it more effectively
- *and* while the model is being constructed so that a user can see what they are doing.

It is facilitated primarily by the separation of the ecological and simulation modelling levels. That separation has been dealt with briefly in chapter 5, we treat it more fully in this chapter.

We have explored the expressive power issue in some detail already; however our treatment is incomplete in two ways. First, we have not yet covered all the types of predication that we require (*e.g.* we have neglected the dialogue level). Second, we have discussed the issue chiefly from a theoretical point of view; we must consider implementation issues. These are covered in this chapter.

The design of the language has been constrained in a fundamental way by the requirement that ultimately, the conceptual distance experienced by end users is minimal. Thus, we have been driven to represent many ecological concepts.

Provision of adequate assistance is multi-faceted. Important issues that we have discussed in this regard include: managing choices (*i.e.* search space identification and control), flexibility, relief from redundant and/or menial tasks, and consistency checking. All of this must be present in the context of an easy to use interface. Much of the necessary support for this is presented in this chapter; the ELK interface is presented in chapter 7.

In this chapter we complete our treatment of the expressive power issue by describing the constructs used in ELK to represent and reason about the variety of predications listed in the beginning of chapter 5. Some will be virtually identical to the object-level constructs presented in chapter 5, others will require elaboration. A key theme will be that of *implicit specification.* By this we mean that a significant portion of the final collective specification of ecological and simulation modelling information will be inferred rather than explicit. The simplest example of this is $\sqsubset^s$ which is the transitive closure of $\sqsubset^s_\circ$. If ELK explicitly represents *lion* $\sqsubset^s_\circ$ *mammal* and *mammal* $\sqsubset^s_\circ$ *mammal*, it will infer *lion* $\sqsubset^s$ *animal* using rule 5.1. The major uses of this are to achieve economy and modifiability. This is traded off against efficiency. In most cases, efficiency has not been an issue. Where it has, some form of compiling has been used. For example, it is not necessary for users to create sorts explicitly. Anything that goes in the $\sqsubset^s_\circ$ or $\prec^p$ hierarchies must necessarily be a sort. It would be possible to not record sorts explicitly, but to infer their existence dynamically, however this is impractical. Instead, ELK infers their existence when users edit the above-mentioned hierarchies and stores them explicitly as Prolog facts. Although the emphasis will be on implementation in this chapter, we will type most of the meta-level constructs introduced. This is in the interest of being precise, and serves as a high level form of program documentation.

We then show how our formalism serves as the foundation for solving the problems of conceptual distance, and model comprehension. In chapter 7 we address the remaining issue of providing assistance during the overall process of acquiring formal descriptions of ecological knowledge, systems, and models.

In doing this, we will be simultaneously testing our ontology completeness and usefulness hypotheses. The first is that all useful information with respect to the process of ecological modelling can be classified in our four-level knowledge ontology. The second is that this is a useful thing to do.

6.2 Introducing ELK

One of the main design constraints is that it must be very easy to modify the general/ecological knowledge base by adding and/or removing sorts, attributes, and part links to suit the specialised needs of individual users. Because we cannot anticipate all the aspects of an arbitrary ecological system *every user will have to play knowledge base designer some of the time.* Therefore, we have designed ELK so that the original knowledge base builders and end users use exactly the same tools for editing the knowledge base. With one exception (noted later) we have succeeded in this aim.

Both object- and meta-level constructs are used by ELK to specify information at all four major information levels in our knowledge ontology. We have not yet covered dialogue level constructs, which are all at the meta-level. We briefly review the constructs that we have seen so far classified into three of the four major information levels:

General/Ecological Level: In editing the sort and/or part hierarchies, users are directly defining the object-level relations $\sqsubset^s_o$ and $\prec^p_o$. Adding to the sort hierarchy also (indirectly) updates the meta-level relations *sort*, and the induced relation *entity_type* which records the valid entity types. For example, to say that *lion* $\sqsubset^s_o$ *mammal* means that *sort*(*lion*) from which we infer *entity_type*(#*lion*). To create attributes we require a special meta-level construct (*att_def*) which specifies the types in addition to various other useful information.

Ecological System Level: In creating entities, specifying their substructure, and indicating what processes they are participating in, users are modifying the description of the ecological system. The creation of entities defines the [system-created] instance relation, '$:_o$'; ':' is implicitly defined (by rule 5.9). Specifying substructure defines $\subset_o$. Specifying that entities participate in processes defines process relations (*e.g. predation*).

Runnable Model Level: By creating model variables users create object-level functions. A construct analogous to *att_def* is used which specifies the function type and contains other useful information.

The meta-level constructs capture a wide variety of information that is required by ELK, but is not represented in the object-level theory. Some are used to define the object-level language, (*e.g.* *att_def*); others are merely used to direct the course of the elicitation process (*e.g.* the dialogue level constructs) and/or to maintain consistency. The constructs presented in this chapter correspond directly to Prolog facts or predicates in the implementation. Most of the time, they are exactly the same. The exceptions correspond either to simplification or rationalisation. Some details are too low-level to be worth including.

Echoing the structure of chapter 2 we first present the details of how to represent the runnable model. We then address the general/ecological knowledge base, the ecological description, and finally the dialogue level constructs. In chapter 7, we show how ELK is actually used.

6.3 Runnable Model

Because the representation of the runnable model is distinct from the representation of the ecological information, it is possible to represent simulation models using *no ecological information at all* (as in figure 2–4). In this section, we give the details of that representation by defining a variety of constructs to represent model variables, initial values, and equations.

6.3.1 Pure versus Ecological

The intention is that these constructs will rarely be used directly. Instead users explicitly associate the simulation model with the description of the ecological system. For example, ELK provides a command which enables a user to create the model variable *n_wb* and explicitly associate it with the attribute *number* of the entity *wb_pop*. The formal connection is made using *attvar_fn* and *effvar_fn*.

ELK also provides a command for choosing equations that apply in specific ecological contexts. For example, to compute the variable *wb_density*, the user may select an equation for computing population density which divides the number

of individuals in the population by the area. The equation itself is of the form $D = N/A$. However, the variables D, N, and A have explicit ecological meaning. Collectively, the equation and the ecological context in which it applies is called an "ecological schema" (details in § 6.4.4).

That using ecological information to construct simulation models facilitates both model comprehension and model construction is the central message of this thesis. The chief technique for facilitating model comprehension is the separation of the ecological and simulation modelling levels in conjunction with explicit association of model components with the ecological system. This enables models to be self-documenting. This separation of ecological and modelling information and bridging the two inherently reduces conceptual distance by allowing ecologists to 'speak' in a familiar language. It also helps define and constrain the modelling search space.

In spite of these benefits, we believe that users should not be forced to make such associations with every aspect of the simulation model. For example it may not be obvious how to do so, or the immediate benefits may be outweighed by the inconvenience. Consider *cf_born* in the example model. It represents the total number of wildebeest calves born in a year. If there is no specific entity corresponding to the wildebeest calves, it is difficult to accurately record the ecological meaning of the variable. To go through the bother of subdividing the wildebeest population just for this purpose might not seem worthwhile. To accommodate such situations we provide a mechanism for defining model variables directly, with no association with ecological concepts. These are called *pure model variables*, they contrast with what we call *ecological model variables.*

We have a similar distinction for schemata. A *pure schema* records no explicit connection with the ecological meaning of the inputs and output(s). For example, the direct proportionality relationship uses the equation $f = a * x$. f, x, and a may correspond to any number of things. The counterpart of a pure schema is an *ecological schema.* For example, the inputs and output for the equation for computing the population density are restricted to specific types with ecological meaning. In particular, the output must correspond to a model variable which represents the attribute *pop_density* of some entity of type *#animal.* There are many advantages of using ecological schemata in preference to pure ones. Chiefly, it helps ensure appropriate use of schemata. For example, ELK would not allow

this schema to be used to compute a variable representing the weight of grass (*grs_wt*). This not only ensures consistency, but restricts the choices faced by the user when selecting schemata.

However, even if we wanted to, it would be very difficult to force users always to use ecological schemata in preference to pure ones. This is because the particular equation that the user requires may not be available in the ecological schema library. A facility for dynamically building up the ecological schema base is difficult to provide, and constitutes an interesting knowledge acquisition project. This will be come clearer in § 6.4.4 when we elaborate on ecological schemata. It is much easier to create pure schemata (dynamically or otherwise). All that is required is to give the inputs and output(s) and the equation. Currently, there are enough schemata to construct the whole of the example model. In the interest of short term expediency, most of them are pure schemata which are simpler to create than ecological schemata (in the current version of ELK). Schemata can only be created manually; however it would be a trivial exercise to implement a facility for dynamic creation of new pure schemata.

Summarising, the representation for the runnable model is distinct from the representation of the ecological information. Users may specify the model directly by creating pure model variables and by using pure schemata. Alternatively, they may specify ecological information first, and then create the runnable model by explicitly associating it with the ecological system being modelled. It is possible to specify the entire runnable model directly, using no ecological information at all. However, this would be inappropriate use of ELK, taking no advantage of its main features; one could be as well or better off using another tool to build models.

6.3.2 Creating Model Variables

Object-level functions corresponding to model variables are created using a series of '*_variable*' meta-level constructs (*e.g. state_variable*, *prate_variable*). There is one for each kind of variable. To define the functions we need to specify two things:

1. its name (*e.g. n_wb*)
2. its range (*e.g. real*)

There are additional bits of information required to specify a model variable fully. The details of what is required varies according to the kind of variable.

- *order:* specifies whether the value space corresponding to the range has the property of being ordered.
- *obtaining values*: this differs according to the kind of variable.
  - *parameter*: a value (*e.g.* $wb\_fec = .5$)
  - *state variable*: an initial value (*e.g.* $n\_wb = 39000$)
  - *partial rate, intermediate,* and *exogenous* variables:
    specification of how the function is computed.
- *increment/decrement information:* (for partial rate variables only) It must be specified which state variable(s) are incremented and/or decremented.
- *text:* a string of text to document the variable. Normally this is automatically generated.

Subsequent values for state variables are computed by integrating differential equations. The ordered property is needed to check whether these functions may be used as arguments to *maximum*, *average*, etc. The system knows that *real* is ordered, but the user has to say whether something like $\{red, green\}$ is ordered.

Values for parameters and initial values for state variables are easily provided. The representation for specifying how to compute intermediate, exogenous, and partial rate variables, is described in § 6.3.4. The following specifications create the variables *n_wb*, *wb_eaten* and *wb_fec* from the example model in chapter 2.

$$state\_variable'(n\_wb, positive, 39000)$$
$$parameter'(wb\_fec, positive, .5)$$
$$prate\_variable'(wb\_eaten, positive, ??, no, n\_wb)$$

The prime indicates that we have omitted the text argument. *positive* is the sort of positive real numbers. The object-level interpretation of this is given below. The fact that *n_wb* is a state variable means that it is computed using a differential equation. This is discussed in § 6.3.4.

$$\begin{aligned} n\_wb &: time \mapsto positive \\ n\_wb(t_{init}) &= 39000 \\ wb\_fec &: positive \\ wb\_fec &= .5 \\ wb\_eaten &: time \mapsto positive \end{aligned}$$

Partial rate variables correspond directly to flows in system dynamics models. The 4th and 5th arguments in the *prate_variable'* construct give the state variables

which are to be incremented and decremented respectively. *wb_eaten* decrements the state variable *n_wb*; '*no*' denotes that no state variable is incremented.

The '??' denotes that the system expects something in that slot; in this case an equation for computing the variable. Normally, the documentation would be automatically generated by ELK, (see § 6.8 for examples).

For conciseness, we shall generally leave out this argument. We also leave out the order argument since this will usually be determined by the type of the range of the function. To type these constructs, we introduce some meta-level types as follows:

- *spec* is the sort of all simple atomic specifications.
- *un_spec* the sort of specifications denoting a form of being unspecified. It is a subsort of *spec*. There are three instances
  - '*na*': not applicable
  - '*no*': could be specified, but is not
  - '*??*': not specified yet, but it should be
- *eqnid*: identifier for some 'black box' which is used to compute the value of the variable, complete with the names of the variables used as inputs.

The first two have no object-level interpretation. We discuss *eqnid* in § 6.3.4. There is a considerable variety of the specification sorts serving different purposes in various contexts. The different kinds are represented as subsorts of *spec*. Their instances constitute the valid inputs to the various 'slots' in the n-ary meta- and object-level relations. Thus ??:*un_spec*, *un_spec* $\sqsubset$ *spec*. Note that in though we are overloading '$\sqsubset$' in this presentation, ELK distinguishes the meta-sort hierarchy from the regular one. The idea of organising these specification types and instances this way is for conceptual clarity and to facilitate consistency checking. It is also used to generate choice menus for these slots.

Note that these meta-types are somewhat different from the meta-types discussed in chapter 5. The earlier ones have direct correspondence to object-level types. For instance we use *variable* as a short hand for the type: (*time* $\mapsto$ *value*) $\sqcup$ *value*. Meta types like *spec* do not have any such correspondence to object-level types. Both kinds of meta types are used to type meta-level constructs. See appendix C for a complete summary of all the meta types and instances.

The types of the *_variable* [meta-level] constructs are:

$$
\begin{array}{ll}
state\_variable' : & variable \times \#value \times value \mapsto bool \\
parameter\_variable' : & variable \times \#value \times value \mapsto bool \\
exogenous\_variable' : & variable \times \#value \times eqnid \mapsto bool \\
intermediate\_variable' : & variable \times \#value \times eqnid \mapsto bool \\
prate\_variable' : & variable \times \#value \times eqnid \times variable^2 \\
 & \mapsto bool
\end{array}
$$

Note that $\#value$ is used as the type for the range of the function. This allows the range to be specified as a sort (*e.g.* $integer{:}set(integer)$), or as a finite set (*e.g.* $\{black, brown, white\} : set(v(colour))$). The range information could be used to spot runtime semantic errors. The values of variables should always stay in the specified range. The value of a variable representing the colour of a sheep should not be red; similarly weight should never go negative.

6.3.3 Pure Model Variables

With respect to the runnable model, the *_variable* constructs just described are sufficient. They carry no ecological information. The *_variable* constructs are implicitly created using explicitly specified pure and ecological variables. There are four constructs which are specified explicitly. *pure_prate_var* is for pure partial rate variables; *pure_model_var* is for all other pure model variables. Correspondingly, $effect\_var$ is used for ecological partial rate variables, and *att_var* is used for all other ecological variables. Ecological variables are discussed in § 6.4.3.

Some examples of creating pure model variables are given below. For all pure variables the user must manually provide text documenting the meaning of the variable. ELK does this automatically for ecological variables.

$pure\_model\_var'(n\_wb, positive, stvar, ??)$
$pure\_model\_var'(n\_pred, positive, parameter, ??)$

$pure\_prate\_var'(wb\_eaten, positive, na, n\_wb)$
$pure\_prate\_var'(bms\_eaten, positive, bms\_wb\_pop, bms\_predators)$

The type of the *pure_model_var* construct is similar to that of the four *_variable* constructs (excluding *prate_variable*). One difference is due to an additional argument for specifying the kind of variable. We use the meta-type *var_spec* for this. It has has 4 instances: *stvar*, *intvar*, *extvar*, and *parameter*. The only other difference is due to the fact that the argument for obtaining values differs

for different types of variables. It may be either a value, or an equation identifier. Formally:

$$pure\_model\_var' : variable \times \#value \times var\_spec \times value \sqcup eqnid \mapsto bool$$

The *pure_model_var* specifications implicitly instantiate one of four *_variable* constructs. The argument for specifying the kind of variable is used to create the appropriate *_variable* construct. The *pure_prate_var* specifications implicitly instantiate the *prate_variable* construct. Users must explicitly give the name of the state variables which get incremented and/or decremented. One argument is for the state variable which increases, one is for the state variable which decreases. We shall see that this is done automatically for ecological partial rate variables. Either but not both of the inc/decrement slots can be '*na*' which denotes: not applicable. If both are filled, this corresponds to the system dynamics case which presumes a transfer of material from one tank to another. This is simulated by increasing and decreasing two different state variables by the same amount. The type of the *pure_prate_var* construct is exactly the same as the *prate_variable* one. We do not repeat it here. In § 6.4.3.3 (page 216) we compare pure and ecological variables.

6.3.4 Equations and Schema

In our example model, and in general, there are two distinctly different kinds of equations: differential, and non-differential. The specification constructs used in ELK for representing these are virtually identical to those used in the original ECO program [Uschold et al, 1986]. The majority of the code in ELK related to equations was directly lifted. We discuss differential equations first.

6.3.4.1 Differential Equations

The differential equations for the simulation model are fully characterised by the *_variable* specifications. They are automatically generated by ELK. Because we use Newton's method for numerical integration, the differential equations are turned into difference equations before the model is run (as noted in § 2.2.3). This works in the following way:

1. There is exactly one differential equation for each distinct *state_variable* specification.

2. This gives rise to a unique net rate variable which is the left hand side of the equation.
   e.g. $rate(n\_wb)$
3. The right hand side of the equation is obtained by adding all the partial rate variables defined by *prate_variable* specifications whose increment argument refers to the state variable, and by subtracting all the partial rate variables defined *prate_variable* specifications whose decrement argument refers to the state variable.

It is a simple matter for the system automatically to create equation 2.10 from the following specifications. That equation is repeated here using $T{:}time$ rather than $yr(I){:}year$ for simplicity.

$$state\_variable'(n\_wb, positive, ??)$$
$$prate\_variable'(wb\_eaten, positive, na, n\_wb)$$
$$prate\_variable'(wb\_die, positive, na, n\_wb)$$
$$prate\_variable'(cf\_surv, positive, n\_wb, na)$$

$$rate(n\_wb)(T) \;=\; -wb\_die(T) - wb\_eaten(T) + cf\_surv(T)$$

Recall that the left hand side represents the net rate of change of the number of wildebeest which decreases due to mortality and predation, but increases due to reproduction. The terms on the right hand side are partial rate variables each representing the contributing effects on the wildebeest population due to one of these processes. Note that we use the atomic variable names rather than the *var_fn* expressions in both the specifications and the equations. This is for readability, both for end users and the reader [of this thesis]. Users cannot be expected to cope directly with expressions like $attvar\_fn(number, wb\_pop)$. The system provides default names for ecological variables (*e.g.* $number\_wb\_pop$); users may override them as desired (*e.g.* n_wb).

6.3.4.2 Non-differential Equations

We require a mechanism for specifying how variables like *wb_eaten*, and *wb_die* are computed. These will in turn depend on other variables and parameters. For example, equation 2.8 indicates that *wb_eaten* depends on *n_pred* and *spr_pred_wb*. The former is constant and thus a parameter in the model. The latter is computed

```
1.  wb_die
2.      n_wb
3.      spr_wb_surv
4.  wb_eaten
5.      n_pred
6.      spr_pred_wb
7.          wb_htime
8.          wb_cap_cf
9.          wb_density
10.             n_wb  (see line 2)
11.             area_sgti
12.         ap_htime
13.         ap_cap_cf
14.         ap_density
15.             n_aprey
16.             area_sgti  (see line 11)
17. wb_repro
18.     spr_cf_surv
19.         grs_wt
20.             dry_ssn_rain
21.     cf_born
22.         wb_fec
23.         n_wb  (see line 2)
```

This computation network for the Serengeti model depicts variable dependency information in a convenient format. It is implicit in the set of equations which constitute the model (see figure 2–4). See figure 6–2 for further details.

Figure 6–1: Serengeti Computation Network

and depends on several other variables corresponding to the attributes handling time, capture coefficient, and density of the wildebeest and aggregate alternate prey populations. The dependency of variables on other variables and parameters constitutes an acyclic graph which we refer to as the computation network. A simplified view of the complete network for the example model is given in figure 6–1. A more detailed view including the equations is given in figure 6–2. Each are examples of ELK output.

Although figures 6–1 and 6–2 give an adequate summary of the state of the specification, there is one major aspect that we have not yet described: reusable schemata which give rise to each equation. For example, there is a commonly used equation used for direct proportionality: $f = a * x$ where a is the constant of proportionality. This is used to compute three of the model variables in the

```
rate(n_wb)(T) = -wb_die(T) - wb_eaten(T) + wb_repro(T)

1.  wb_die                          = n_wb* (1-spr_wb_surv)
2.      n_wb (SV)
3.      spr_wb_surv = 0.95

4.  wb_eaten                        = n_pred*spr_pred_wb
5.      n_pred = 1200
6.      spr_pred_wb         = wb_cap_cf*wb_density/
                               (1+ (wb_htime*wb_cap_cf*wb_density +
                                    ap_htime*ap_cap_cf*ap_density))
7.          wb_htime = 0.08
8.          wb_cap_cf = 317
9.          wb_density              = n_wb/area_sgti
10.             n_wb (SV)
11.             area_sgti = 1000000
12.         ap_htime = 0.05
13.         ap_cap_cf = 100
14.         ap_density              = n_aprey/area_sgti
15.             n_aprey = 4200
16.             area_sgti = 1000000

17. wb_repro                        = cf_born*spr_cf_surv
18.     spr_cf_surv                 = (grs_wt+0.05)/ (75+grs_wt)
19.         grs_wt                  = 200+dry_ssn_rain*2
20.             dry_ssn_rain = 250
21.     cf_born                     = n_wb*wb_fec
22.         wb_fec = 0.5
23.         n_wb (SV)
```

*This shows the differential and non-differential equations which constitute the example model of the Serengeti (see figure 2–4). The latter give rise to an acyclic directed graph. The root nodes correspond to partial rate variables, or possibly intermediate variables which are not used to compute anything, but are of interest as an output of the model (*e.g. *an average over the course of the simulation). There are two kind of leaf nodes, state variables (*e.g. *n_wb) and parameters (*e.g. *n_pred). This is a more verbose version of figure 6–1.*

Figure 6–2: Serengeti Equations

example (*wb_eaten*, *cf_born*, and *wb_repro*). A schema has a number of inputs and outputs, and an arbitrarily complex procedure for computing the outputs from the inputs. In our example, these procedures are fairly simple equations. Also, our schemata only have one output. For example, the schema for direct proportionality is defined as follows:

pure_schema(*dirp*, $\{x, a\}$, $\{f\}$, $f = x * a$, 'Direct Proportional Relationship').

A schema is instantiated by assigning a model variable to each input and output. To distinguish it from other uses of the same schema, we give it a unique name. This schema is instantiated to compute *wb_eaten* as follows. A unique name is generated (*dirp/3*); the input a is assigned to *spr_pred_wb*, and x to *n_pred*. This is represented as follows (using Prolog terms):

Instantiated schema identifier:	`ischema('dirp/3',dirp,[x,a])`
Attach output to variable	`outarc(wb_eaten,'dirp/3',f)`
Unique input identifiers:	`inarc('x.5','dirp/3',x)`
	`inarc('a.4','dirp/3',a)`
Attach variables to input identifiers:	`tie('a.4',spr_pred_wb)`
	`tie('x.5',n_pred)`

Note that each schema has a unique identifier for each input. The variables that these correspond to are specified using the *tie* construct. This is somewhat more complicated than it might be to facilitate maximum flexibility in changing inputs and outputs. For example, if it turns out that the variable for the a input should be different, only the *tie* specification need be altered. Other examples explaining the flexibility of this representation are described in [Uschold et al, 1986]. This is not central to our concerns, nor does ELK currently make full use of the flexibility afforded by this representation. We omit typing these constructs.

At this point, we can say exactly what goes in the *eqnid* slot in the *_variable* constructs for intermediate, exogenous, and partial rate variables. It is the instantiated schema identifiers (*e.g. dirp/3*). In conjunction with the above constructs, this identifier characterises how to compute the variable.

Logically, each schema is a function of type $value^n \mapsto value$. These functions correspond to procedures in conventional programming languages (subroutines in Fortran). An instantiated schema corresponds to an expression of type *value* (usually *real*) with unbound logical variables for time. For example, the above specification says that:

$$\begin{aligned} \forall T : time.wb\_eaten(T) &= dirp(n\_pred, spr\_pred\_wb(T)) \\ &= n\_pred * spr\_pred\_wb(T) \end{aligned}$$

6.3.5 Running the Simulation

We have now described the formal representation for runnable models and shown how various constructs may be used to define model variables and differential and non-differential equations. To run a model, all parameters and state variables must be initialised and the differential equations numerically integrated. Additionally, users must specify which variables they wish to output, and how frequently.

The program generation component of ELK is crude and intended for demonstrative purposes only. Functionally, it amounts to re-implementation of the code generator in ECO [Uschold et al, 1986]; however the simulation is run in Prolog, not Fortran. To do this, we simply added assignment and looping constructs and made Prolog behave like Fortran (consequently, the simulations run very slowly). Other than short term expedience, we do not claim any advantages of this approach over generating and running Fortran code.

When the user issues a compile command, the computation network is topologically sorted. Each node gives rise to a Prolog goal isomorphic to what would be a subroutine call in Fortran. These are placed in a loop exactly as they would in a Fortran program. The compiler detects intermediate variables that do not belong in the main loop because they are constant (*e.g.* *spr_cf_surv*, and *grs_wt*). These depend ultimately only on parameters, not on state or exogenous variables. They are computed once only, before the main simulation loop is executed. These belong to the third point along the continuum of variability discussed in § 2.4.1.1 (page 47); this is the only time that we distinguish these intermediate variables from other proper model variables. The execution of the simulation is isomorphic to that of the Fortran program that the old ECO would have generated.

The simulation may be run over and over with different values for state variables and parameters. Of course, if the computation network changes in any other way (*e.g.* adding a variable, changing the input to an instantiated schema), it must be recompiled. Currently it is the user's responsibility to recompile the model if they change it; it would be a minor task to enhance ELK so that it checks for this. Another frill that could be easily added is to do run-time consistency checking to make sure value spaces are not violated. For example, the number of wildebeest

could conceivably go negative. All variables whose value spaces were constrained in some way and could be violated should be checked for this at run time. This is an example of how having an explicit ecological account of the model can be useful.

It is evident from the above discussion that the programs currently generated could trivially be translated to Fortran and run efficiently. However, there is a restricted class of models for which this is currently possible. We have not explored in detail what the limitations are, however it is likely that complex substructure will require something fairly clever. Although, this is not central to our concerns, we recognise that it is an important problem that will need to be solved before programs generated using an ELK-like approach can be used in anger. Some preliminary work has been done by others in our research group in converting models with substructure similar to that described § 5.6.6.3 into Basic.[1]

As it stands, the constructs used to represent the runnable model can represent differential (and difference) equations models which incorporates the system dynamics framework. As a target language for runnable simulation models it is sufficiently powerful to represent a wide range of useful models, and is thus entirely adequate. That portion of ELK which may be used to build and run simulation models using the runnable-model constructs *only* offers no significant advantages over tools like ECO [Uschold et al, 1989] or Stella [Lewis, 1986]. We need to go beyond the runnable-model level to meet our requirements from chapter 4; these are:

- to facilitate model comprehension
- to reduce conceptual distance
- to assist in the identification of the idealisation search space
- to ensure ecological consistency checking
- to provide relief from menial and/or redundant tasks

To meet these requirements, we must represent ecological information. This is discussed in the next section. This gives rise to a greatly enhanced modelling tool which far surpasses that of ECO and more conventional tools like Stella. The improvement will derive both from:

---

[1] Robert Muetzelfeldt, personal communication.

1. increased ease with which models may be
   - comprehended (by recording ecological meaning of model elements)
   - constructed (by reducing conceptual distance)
2. the ability to build more complex models

The second point is important, but the emphasis in this thesis is much more on model comprehension and construction. To some extent the extra complexity of the models is not due to a more expressive runnable-model language, but the existence of high-level constructs (*e.g.* substructure, aggregation operators like *average*) that can be compiled into this language.

6.4 Ecological Information

In this section, we explore how the representation for the example model in chapter 2 is augmented to include the necessary ecological information. Although, ultimately the final model may appear exactly as it did in chapter 2 variable names and all, the difference is that the variable names link into a network of ecological information that will have been used in the construction of and can be used to explain the model.

We have already discussed how much of the ecological information we require is represented. We have concentrated on describing the object-level language. We have skimmed over the sections on attributes and processes which require meta-level constructs to be specified. In the following sections we summarise the material already presented and elaborate as appropriate. All the examples are based on what is required to describe the ecological and simulation modelling information for specifying the simple Serengeti model. Later, we consider briefly how this may be extended and the usefulness of doing so.

6.4.1 General/Ecological Knowledge

The general/ecological knowledge base contains the following:

- various general purpose functions and relations:
  e.g. *set*, $\sqcup$, *maximum*, *qnam_ent*
- a hierarchy of types of entities induced from a hierarchy of sorts:
  e.g. $\sqsubset$, $\sqsubset^s$

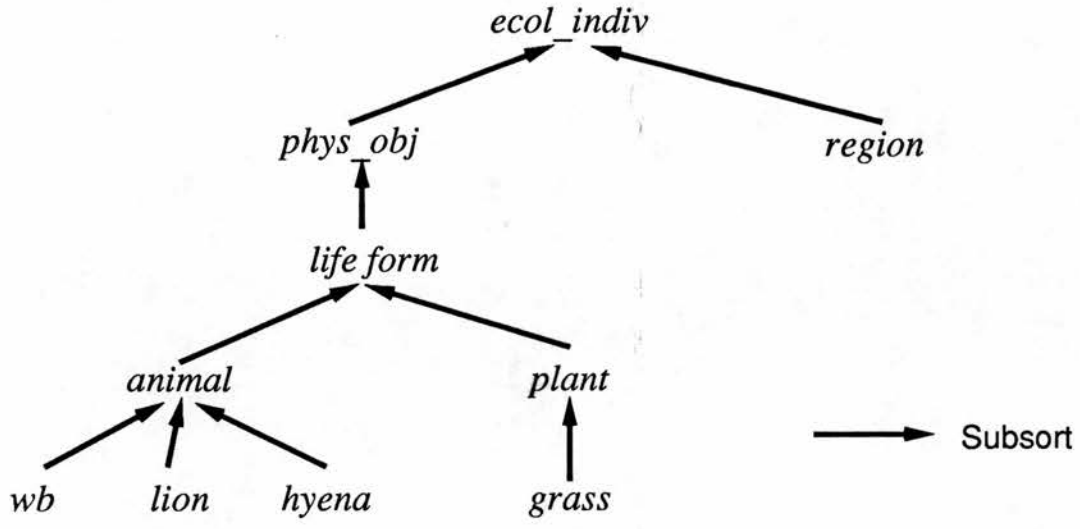


Figure 6–3: Serengeti Sort Hierarchy

- what sorts of entities can be composed of what other sorts of entities:
  e.g. $\prec^p$, $\subset^p$
- information about the attributes that apply to these sorts of entities
  e.g. *number* applies to entities of type $set(entity)$.
- information about processes that these sorts of entities participate in that affect the attributes.
  e.g. *animals* prey on *animals*

6.4.1.1 Types, Substructure

The following sorts of ecological entities are adequate for the basic needs of the simple Serengeti model: *ecosystem*, *phys_obj*, $lifeform$, *animal*, *plant*, *wb*, *lion*, *hyena*, and *grass*. We also make use of the induced collection sorts, $\#wb$, $\#lion$, $\#hyena$, and $\#animal$. These are organised in a simple sort hierarchy illustrated in figure 6–3.

There is no part-composite substructure required, thus the possible part hierarchy ($\prec^p$) is not used. The induced possible component relation permits the set substructure that we require. In particular, $\#lion \subset^p \#animal$ and $\#hyena \subset^p \#animal$ permit us to explicitly create the aggregate predator population.

6.4.1.2 Attributes

To fully specify the general/ecological knowledge regarding an attribute, we require the following:

- A: the [name of the] attribute (*e.g.* sex)
- T: the entity type to which it applies (*e.g.* $lifeform$)
- Vs: the value space defining its range (*e.g.* $\{male, female\}$)
- DVs: a dimension value space if applicable (*e.g.* $\{male, female\}$)
- Vb: a variability specification (*e.g.* $constant$)
- Val: a default value if applicable (*e.g.* na[2])

For finite value spaces, the user must also indicate whether the values are ordered or not. The system knows this already for number sorts like *real*, and *positive*. Because order is a property of value spaces, not of attributes, we omit the order specification in this discussion.

The dimension value space is a finite set of values which is used to define substructure for entities of a specific type. These represent equivalence classes on the value space. For example for the attribute weight, *small* might be equal to the range from 0 to 10 (for some unit). It is unusual for this to be appropriate to be defined at the general ecological level since these classes will rarely be universally recognised as standard. A counter example is given in the example above for sex; another might be the age categories for lion $\{cub, adult\}$.

The variability time scale is an indication of on what time scale the value of the attribute for that type changes. Currently we just have two points on this scale; *constant* denotes an effectively infinite time scale; *variable* everything else. More points on this scale (*e.g.* *year*, *week*) could be used in conjunction with the specified time scale of the model to suggest whether an attribute should be modelled as a parameter or a constant. For instance, capture coefficient for a population will not ordinarily change. For variables, the default value slot is not applicable.

Some attributes have generally applicable default values. For instance, the fecundity of wildebeest populations may be the same the world around. A default value would denote the average fecundity of all wildebeest. If there are

---

[2] Recall, this is for not applicable.

useful default values which do not hold generally, these do not belong in the general/ecological knowledge base. In § 6.5.3, we describe a separate mechanism to handle that case.

The construct used to create attributes is *att_def* (we saw a shortened version on page 179, chapter 5). To type this we require one more meta-type, *vb* for variability. Its instances are *constant* and *variable*. Below we give the type of *att_def* and an example showing how to create the attribute weight. For the moment, ignore the '$_b$' subscript.

$$att\_def : \quad attribute \times \mathcal{P}(ecol\_ent) \times (\#value)^2 \times (vb \sqcup time) \times value \mapsto bool$$
$$att\_def_b(weight, phys\_obj, positive, na, variable, na)$$

The first two arguments identify this particular specification, the latter four collectively constitute what we shall refer to as the *attribute description.*

The *att_def* constructs are used to create and characterise the ecological attributes. Formally, they are meta-level constructs which specify the types of object-level functions corresponding to attributes. These functions describe the state of the ecological system being modelled. They are distinct from the functions corresponding to the model variables which idealise them. The attribute functions themselves will not usually be defined (*i.e.* the actual n-tuples) except in a very limited sense where field data is available. The object-level interpretation of the above *att_def* specification is to create the function *weight* as follows:

$$weight : phys\_obj \times time \quad \mapsto \quad positive$$

If $yr80$ represents 1980, then $weight(grass, yr80) = 200$ says the weight of the grass in 1980 was 200. This could be real field data. How and whether the years are represented in the description of the ecological system, (*i.e.* distinct from the simulation model), depends on the user's needs and perhaps the nature of the available data. Currently, we provide no support for having two separate time lines, one for the real world and one for the simulation.

Sometimes default values and/or dimension value spaces are applicable to subsorts of the most general sort to which an attribute applies. For example, we noted above that that *lamb* and *ewe* are values that are generally applicable for sheep. We use the construct att_def_b for the specification of an attribute of the most general sort. This is the basic case. We use att_def_m for modified versions. This is illustrated below:

$$att\_def_b(age, phys\_obj, positive, na, variable, na)$$
$$att\_def_m(age, sheep, positive, \{lamb, ewe\}, variable, na)$$

The complete interpretation of the first is that *age* is an attribute of entities of the sort: *phys_obj* (and all subsorts); possible values are positive real numbers; it varies, and it is not meaningful to assign a default value, or a dimension value space. The second is almost the same except that a generally useful set of values for defining age substructure is $\{lamb, ewe\}$. The practical implication of this specification is that if age substructure is ever defined on any sheep when the ecological system is being described, the user should be offered these values first.

Note that even for sheep, it still makes no sense to speak of a default age. However we might specify a default value for the attribute *weight* with respect to *sheep* (but not *phys_obj*). Users may not change the value space type, in the modified versions. This is because the range of of the function *weight* would then change and *weight* would be ambiguously typed.

When the value spaces are finite sets (*e.g.* colour) the value space specification in the aff_def_m constructs may differ, however this would not change the type. For example, the attribute *colour* might have $\{brown, white\}$ for the sort *lion*, but $\{green, yellow, red\}$ for the sort *apple*. ELK keeps track of all the colours specified for any sort and these automatically become instances of $v(colour)$. Because is is too expensive to chase up and down all the att_def constructs every time the system needs to check the type of an instance, these values for qualitative sorts are stored separately. This is another example of where the user sees the effect of implicit specification, but underneath there is a compilation going on. Redundancy is avoided by only adding new colours to the set. The construct used is *qualitative*. For example, the above two specifications would result in the following specification:

$$qualitative(colour, \{black, white, red, yellow, green\}).$$

which records the fact that there are five values of the attribute *colour* that the system knows about. The user may also add colours explicitly by creating instances of type $v(colour)$. The abstract and full types of *qualitative* are given below. N.B. we use *set* here, not # signifying that the set of values must be flat.

$$\begin{aligned} qualitative: \quad & attribute \times set(value) && \mapsto bool \\ & (ecol\_ent \times time \mapsto V) \times set(V) && \mapsto bool \end{aligned}$$

In § 5.6.6.3 (page 167) we hinted at another form of redundancy that could arise in this context. If a user wished to create the attributes *eye_colour* and *skin_colour* we do not want also to have the sorts $v(eye\_colour)$ and $v(skin\_colour)$. The current implementation does not handle this problem, but we outline a simple method for doing so. The idea is to use indirect reference to specify "use the same value space as another attribute". If we used the symbol '@' for this the user would enter *@colour* in the value space slot for *skin_colour*. This would define *skin_colour* as a function from *animal* and *time* to $v(colour)$. The specific colours that apply to different sorts of animals may be specified in the usual way. There would still be a single qualitative specification that keeps track of all colours. This requires minimal user effort.

Inherited and Induced Attributes

To simplify the following discussion, we introduce some shorthand notation for the 6-ary attribute relations. For each attribute specification, the attribute name and the type to which it applies serve to uniquely identify it. The other parts, Vs, Dvs, Vb, and Val are collectively referred to as the attribute description. We use a ternary version of these relations which merely packages up the attribute description into one unit. It will appear as D in the formulae. We introduce the type abbreviation *attdesc* to refer to 4-tuples as follows:

$$\begin{array}{lcl} D{:}attdesc & \rightarrow & D{:}(\#value)^2 \times vb \times value \\ att\_def(A,T,(Vs,Dvs,Vb,Val)) & \leftrightarrow & att\_def(A,T,Vs,Dvs,Vb,Val) \end{array}$$

Both att_def_b and att_def_m are directly specified by the user rather than inherited or induced. We use $att\_def_\circ$ to blur the distinction between these two. We define the relation att_def in terms of $att\_def_\circ$. It incorporates automatic inheritance of attributes from sorts to subsorts, as well as induced attributes using *average* etc.. For example, *lion* inherits the attribute *weight* from *phys_obj*; sets of physical objects 'inherit' the induced attribute $qnam\_ent(maximum, weight)$. The relationship between $att\_def_{b/m}$ and att_def is similar to that between $\sqsubset^s_\circ$ & $\sqsubset^s$, $\subset_\circ$ & $\subset$, etc. In each case, the latter is derived from the former using inference. Formally:

$\forall A{:}attribute.\forall Q{:}squal.\forall T, T_1, T_2 \sqsubset ecol\_ent.\forall D, D', D''{:}attdesc$

$$
\begin{array}{lcl}
att\_def_b(A,T,D) & \rightarrow & att\_def_o(A,T,D) \\
att\_def_m(A,T,D) & \rightarrow & att\_def_o(A,T,D) \\
att\_def_o(A,T,D) & \rightarrow & att\_def(A,T,D) \\
att\_def_o(A,T_2,D) \wedge T_1 \sqsubset T_2 & \rightarrow & att\_def(A,T_1,D) \\
att\_def(A,T,D) & \rightarrow & att\_def(qnam\_tim(Q,A),T,D') \\
att\_def(A,T,D) & \rightarrow & att\_def(qnam\_ent(Q,A),set(T),D'')
\end{array}
$$

For example:

$\forall D, D', D''{:}attdesc$

$$
\begin{array}{lcl}
att\_def(weight, phsy\_obj, D) & & \\
\quad \wedge sheep \sqsubset phys\_obj & \rightarrow & att\_def(weight, sheep, D) \\
att\_def(weight, phys\_obj, D) & \rightarrow & att\_def(qnam\_tim(total, weight), phys\_obj, D') \\
att\_def(weight, phys\_obj, D) & \rightarrow & att\_def(qnam\_ent(total, weight), set(phys\_obj), \\
 & & D'')
\end{array}
$$

Note that D' is related to D. The value spaces will usually be the same, but if the value space for an attribute is integers (*e.g.* *number*), then the value space for $Qnam(average, number)$ will be *positive* because the integers are not closed under division (where $Qnam$ is either *qnam_ent* or *qnam_tim*). The dimension value space and variability always stay the same. For average only, the default value remains the same, otherwise it always changes to or remains as not applicable. Another frill which would be easy to implement would be to compute a sensible default value for total (but not maximum or minimum) if there is a default value for *number* of entities in a set of something. For example, if the default number of lions in a pride was 10, and the default weight for a lion was 120, then the default value for total weight of a set of lions would be 1200.

Two other things to note are 1) that in all cases the value space must be ordered, and 2) if $Q = average$ addition and division must be defined on (but not necessarily closed under) the value space. ELK checks for this.

The definitions also give us the ability to induce more complex attributes with nesting. If we apply the rule above for *qnam_ent* where
$A = qnam\_tim(average,weight)$, and $Q = maximum$ we induce the attribute:
$qnam\_ent(maximum,qnam\_tim(average,weight))$ which applies to sets of physical objects. If the average weight of each of a number of objects was computed

for some set of times, this attribute represents the maximum of these averages for that set of objects. Although these may be nested indefinitely, in practice, it is hard to imagine more than two or three levels being required except on very rare occasions.

Elk computes the full new types of the induced attributes (*i.e.* by nesting *set* as required, not using #. It also computes the full types of collections when they are used as arguments to any variables that induced attributes give rise to. For example, $qnam\_ent(maximum, qnam\_ent(average, weight))$ applies to entities of $set(set(phys\_obj^{\odot}))$. Thus, if this is to be computed for $pride{:}\#lion$, ELK must first check that *pride* has the appropriate substructure defined.

Note that formally, these induced attributes are exactly like any others. They may give rise to their own model variables. Their value spaces may be partitioned and used as dimensions for defining substructure, etc. In practice, they will tend to be used in special ways, but this is purely an 'accident' of what modellers happen to find useful to do.

6.4.1.3 Processes

Processes are the most complex concept that we represent. We glossed over some important details in § 5.4, mostly because we do not attempt to capture everything in the object-level logic. Here we elaborate. There are two aspects to defining processes. One is to say how many and what kinds of entities may participate. The other is to say what attributes are affected by these processes and how.

Each process may have 1 or more participants (usually not more than 2). We shall refer to the participating entities as agents. For processes with 2 or more agents, each agent is in a specific role. For instance, in the process of predation one entity is in the *predator* role and the other is in a *prey* role.

We use the construct $role\_def$ to define process roles. There is not always a useful label to attach to a role. In this case, we merely use *agent*. We require three things to specify a role for a process:

- The name of a process (*e.g. predation*)
- The name of the role (*e.g. predator*)
- The type of the entity that may play that role in the process (*e.g.* $\#animal$)

For example, the following specifies the two roles for the predation process, and the single role for the mortality process.

$$role\_def(predation, predator, \#animal)$$
$$role\_def(predation, prey, \#animal)$$
$$role\_def(mortality, agent, \#lifeform)$$

Processes only affect certain attributes of the participating entities. These are associated with what we call effects. Each effect requires the following to be fully specified:

- The name of a process (*e.g. predation*)
- The name of an attribute (*e.g. amount_ddt*)
- One or two roles which are affected (*e.g.* $\{prey\}$)
- The nature of the effect (*e.g. decrease*)
- The time scale which is an indication of the minimum time period during which such effects are measurable (*e.g. second*)
- Whether it necessarily or possibly is associated with each occurrence of the process. (*e.g. possible*)

Some examples are given below. Note that at the general/ecological level, the $transfer$ option means that *in reality* the effect causes a transfer of the 'stuff' represented by the attribute. This specification at this level means that by default, the effect will be modelled as a system dynamics flow.

$$effect\_def(predation, number, prey, decrease, second, necessary)$$
$$effect\_def(predation, weight, \{prey, predator\}, transfer, second, necessary)$$
$$effect\_def(predation, amount\_ddt, \{predator, prey\}, transfer, second, possible)$$

To give the types for these meta-level constructs, we need first to introduce some more meta-types. With the exception of the first one which results in dynamically created instances, all of these sorts and instances are permanent.

- *role*: the sort of all roles of processes (*e.g.* $grazer : role$)
- *chg_spec* is the sort of change specifications for effects. There are exactly four instances, two basic ones ($transfer$, and *change*), and two inherited from $direction\_spec \sqsubset chg\_spec$ (see immediately below).
- *direction_spec*: the sort of all direction specifications. There are exactly two instances, both basic: (*increase* and *decrease*).
- *necpos*: the sort of necessity specifications. It's two instances are *necessary* and *possible*.

$$role\_def : process \times ecol\_ent \times agency \mapsto bool$$
$$effect\_def : process \times attribute \times role^{\odot} \times chg\_spec \times time \times necpos \mapsto bool$$

The set of n $role\_def$ specifications associated with a process creates an object-level n-ary relation which says which entities are involved in that specific process. For example the $role\_def$ specifications above create the object-level relation:

$$predation : \#animal \times \#animal \mapsto bool$$

Each $effect\_def$ specification creates a function which corresponds to the 'real' (*i.e.* not idealised) rate of change of the attribute(s) of the entity(ies) participating in the process. The object-level representation of this function is more difficult to capture that it was for attributes. We defer the discussion of this to § 6.4.2.

6.4.2 Ecological System Description

The description of the ecological system consists of:

- specific entities (*e.g.* $ln\_pop : \#lion$),
- substructure of these entities (*e.g.* $ln\_pop \subset predators$)
- attributes of these entities
  e.g. *number* is an attribute of *ln_pop*
- the processes that the entities are participating in and their effects
  e.g. *predators* prey on the wildebeest population (*wb_pop*) thereby tending to decrease their numbers.

6.4.2.1 Entities

The specific entities that are part of the Serengeti that we are interested in include several animal populations (wildebeest, lions, hyena, etc) and grass. There is sub-structure in the form of aggregated populations. The lion and hyena populations are aggregated into a single predator population. Similarly, the alternate prey population consists of several populations of different species (we ignore these for now). Formally:

$grs{:}grass$ $\quad$ $wb\_pop{:}\#wb$ $\quad$ $ln\_pop{:}\#lion$
$hyena\_pop{:}\#hyena$ $\quad$ $predators{:}\#animal$ $\quad$ $alt\_prey{:}\#animal$

$ln\_pop \subset predators$ $\quad$ $hyena\_pop \subset predators$

The implementation has a relation called *base_inst* which corresponds closely (but not exactly) to '$:_\circ$'. For primitive entities, they are the same. The difference is that we allow $base\_inst(ln\_pop, \#lion)$, but not $ln\_pop :_\circ \#lion$ because the least type of something can never be of the form $\#T$. As noted in chapter 5, we never need to know the least type of collections. The only time we need to know anything more about the type than the fact that it is a collection, is if we need to compute maximum or average or some nested combination of these. For example, if

$$\begin{array}{ll} wb(1), wb(2), wb(3), wb(4) : wb & wb\_pop, wb\_subpop : \#wb \\ wb(1), wb(2), wb\_subpop \subset_\circ wb\_pop & wb(3), wb(4) \subset_\circ wb\_subpop \end{array}$$

then the substructure of *wb_pop* is depicted in the set $\{wb(1), wb(2), \{wb(3), wb(4)\}\}$. It makes sense to compute average weight of *wb_pop*; we need only flatten the set. So,

$$qnam\_ent(average, weight)(wb\_pop, T) = (\sum_{i=1}^{4} weight(wb(i), T))/4$$

However, it does not make sense to compute maximum average because *wb_pop* is not a set of sets. The general rule is that if maximum, average, etc are nested i levels deep, then it is possible to fully perform the computation if and only if the collection over which the computation is to take place is an instance of $set^{(i-1)}(\#S)$. Recall from figure 5–1 that $set^{(0)}(S) = S$, $set^{(2)}(S) = set(set(S))$, etc. The precise type of *wb_pop* is $set(wb \sqcup set(wb))$ which is a subtype of $set^{(1-1)}(\#S) = \#S$, but not $set^{(2-1)}(\#S) = set(\#S)$.

We require a special mechanism for dealing with categories used in dimension value spaces. For example, the dimension value space for *age* of *lion* might be $\{cub, adult\}$. Currently, ELK checks such entries and automatically creates these as instances of $\#V$, where V is the value space of the attribute (in this case, *positive*). Users may or may not choose to explicitly define these sets. For qualitative sorts, they could do so in the usual way using the $\subset_\circ$ relation. For example, $dark : \#v(colour)$ might be a category for *colour*, its components being *purple*, *blue*, *black* etc. For continuous sorts like numbers, this is impractical. We therefore provide a facility for defining intervals. For example, $interval(cub, [0, 1])$ specifies that *cub* is identified with the interval $[0, 1]$.

6.4.2.2 Attributes

The fact that an attribute applies to a specific entity cannot be viewed as general/ecological knowledge, because the entities are specific to the ecological system being modelled. We have a separate construct called *att_esys* for recording this. It is defined as follows:

$$att\_esys : attribute \times ecol\_ent \times attdesc \mapsto bool$$

$$\forall A{:}attribute.\forall T \sqsubset ecol\_ent.\forall E{:}T.\forall D{:}attdesc.$$
$$att\_def(A,T,D) \rightarrow att\_esys(A,E,D)$$

Each entity (*e.g. grs*) inherits all the attributes that its type has, which are either directly specified by the user (via att_def_0) or are inherited from the supersorts (*e.g. plant*, *phys_obj*, etc.), or are induced using function qualifiers. Thus, each population has the *number* attribute, and the grass entity has the attribute *weight*. These entities also inherit the induced attribute 'maximum number over some period of time' (*i.e.* $qnam\_tim(maximum, number)$). For example:[3]

$$att\_esys'(weight, grs);\ att\_esys'(number, wb\_pop)$$
$$att\_esys'(number, predators);\ att\_esys'(number, alt\_prey)$$
$$att\_esys'(qnam\_tim(maximum, number), wb\_pop)$$
$$att\_esys'(qnam\_ent(total, weight), wb\_pop)$$

Note that all of this is implicitly specified. It is available for querying the description of the ecological system, should that be necessary.

6.4.2.3 Processes

Of the various processes going on, we shall limit this discussion to the occurrences of predation between the aggregate predator population and the two prey populations. The latter are the wildebeest population and the aggregate alternate prey population. The necessary effects of processes are automatically inherited. Thus, the *number* attribute of the prey population necessarily decreases; similarly, the attribute *weight* of the predator and prey populations increases and decreases

---

[3] Recall that the prime denotes that a simplified version of the actual construct is being used.

respectively. The user must specify whether the optional effects are occurring in the ecological system being modelled (in this case they do not).

Before we present the meta-level construct for specifying processes at the ecological system level, we first consider the object-level interpretation of processes. Ultimately, we are interested in modelling the effects of processes. To characterise a single effect uniquely, we require four things:

- the name of the process (*e.g. predation*)
- the participating entities and their corresponding roles (*e.g.* $(predator, predators)$ and $(prey, wb\_pop)$).
- the affected attribute (*e.g. number*)
- the affected entity or entities (*e.g. wb_pop*)

Because the first two will remain constant for all the effects for a particular occurrence of a process, we introduce the meta-level type: *occ*, which is a type of 2-tuple. The first item in the tuple is a process; the second is the set of the participating entities and their corresponding roles. It will be convenient to give the occurrences names which correspond to instances of this type. For example, $wb\_pred : occ$ uniquely identifies an occurrence of the predation process, the one between the predator population and the wildebeest population (*i.e. predators* and *wb_pop*). Formally:

$$\begin{aligned} O{:}occ &\rightarrow O : process \times set(role \times ecol\_ent)) \\ wb\_pred &= (predation, \{(predator, predators), (prey, wb\_pop)\}) \end{aligned}$$

Recall that we distinguish between the real (or hypothetical) effect and how it is idealised in the simulation model. For example, the real effect of predation is to reduce the number of prey, but in the simulation, this effect may or may not be modelled this way, or it could be ignored entirely. In the example model, the effect on the aggregate alternate prey population is indeed ignored, because the *n_aprey* is held constant. The effect of predation on the wildebeest population is modelled explicitly (see figure 2–4). This distinction between real and idealised effects enables ELK to explicitly represent these important modelling decisions. We represent the real effect using the higher-order function $effect\_fn$. In chapter 5, we referred to the real effect which reduced the number of wildebeest due to predation as wb_eaten', and the idealised version as wb_eaten. Now we can capture the connection between these names and the underlying ecological processes that

they represent. The abbreviated and full types of $effect\_fn$ are given below with examples.

$$\begin{array}{l}
\forall T \sqsubset ecol_ent.\forall V{:}value.\forall n{:}natural. \\
effect_fn \;:\; occ \times attribute \times ecol_ent \mapsto effect \\
effect_fn \;:\; ((T^n \mapsto bool) \times set(role \times T)) \times (T \times time \mapsto V) \times T \\
\qquad\qquad \mapsto (time \mapsto V)
\end{array}$$

e.g.

$$\begin{array}{lcl}
wb_eaten' & = & effect_fn(wb_pred, number, wb_pop) \\
wb_eaten & = & effvar_fn(effect_fn(wb_pred, number, wb_pop))
\end{array}$$

Note that T and V each occur more than once in the full type of $effect\_fn$. This formally captures the intimate relationship between attribute and processes. This completes the discussion of the object-level interpretation of processes. We now continue with the implementation issues.

To identify an occurrence of a process uniquely requires specifying the kind of process (*e.g.* predation) and specifying what objects are participating in what roles. To specify an occurrence fully, we must know what effects are occurring. The necessary ones come for free, the optional ones must be explicitly noted. We use the constructs $role\_esys$ and $effect\_esys$ to represent occurrences of processes and their effects. For example:

$$\begin{array}{lll}
role_esys: & occ \times role \times ecol_ent & \mapsto bool \\
effect_esys: & occ \times attribute \times role \times time \times chg_spec & \mapsto bool
\end{array}$$

e.g.

$role\_esys(wb\_pred, prey, wb\_pop)$

$role\_esys(wb\_pred, predator, predators)$

$effect\_esys'(wb\_pred, number, prey, decrease)$

$effect\_esys'(wb\_pred, amount\_ddt, [predator, prey], transfer)$

wb_pred denotes a single occurrence of the predation process, the one between the predator population and the wildebeest population.

The definition of $effect\_esys$ is given below. It is defined implicitly in terms of an explicit version, $effect\_esys_\circ$ in the usual way.

$$\begin{array}{l}
\forall O{:}occ.\forall A{:}attribute.\forall R{:}role^{\odot}.\forall E{:}ecol_ent.\forall C{:}chg_spec.\forall T{:}time. \\
effect_esys_\circ(O, A, R, C, T) \rightarrow effect_esys(O, A, R, C, T) \\
effect_def(P, A, R, C, T, necessary) \\
\quad \wedge\, role_esys(O, R, E) \rightarrow effect_esys(O, A, R, C, T)
\end{array}$$

where $O = (P, \{...\})$

The object-level interpretation of the *role_esys* specifications is to add two 2-tuples to the predation relation. That is:

$$\begin{aligned} predation(predators, wb\_pop) &= true \\ predation(predators, alt\_prey) &= true \end{aligned}$$

The object-level interpretation of a single instantiated $effect\_esys$ construct is to create a function represented using $effect\_fn$ as discussed above.

We *do not* capture the meaning of the change specification at the object level. This turns out to be difficult to do anyway. *in/decrease* cannot be formalised in the obvious way, since the interpretation here is that the effect *tends* to in/decrease the value of the attribute. It is a local effect, and only partially determines whether there is an global in/decrease. The direct meaning of *decrease* is restricted to the ecological systel level. However, it plays a role in guiding and constraining the specification of the runnable model. By default, a variable representing this effect will be a partial rate variable which is a negative term in the right hand side of the differential equation. It may not be a positive term, because that would make no ecological sense. Of course, the effect may not be modelled at all; if it is, it need not be idealised as a partial rate variable.

6.4.3 Ecological Model Variables

In § 5.7 we introduced the higher-order functions $attvar\_fn$, $attparm\_fn$, and $effvar\_fn$ which are used to represent ecological variables. However, the functions corresponding to variables must be created just as the functions corresponding to attributes and effects must be created. One of the reasons for separating the ecological information from the simulation model is due to the fact that although any given entity is likely to have up to a few dozen attributes (as per *att_esys*), only a very small number will be used explicitly in the simulation model. We do not want to have dozens of variables around that never get used.

Analogous to the *_def* meta-level constructs for creating ecological attributes and effects, we have *_var* meta-level constructs for creating ecological model variables. *att_var* is used to create all ecological model variables corresponding to attributes. This excludes partial rate variables. $effect\_var$ is used to create effect variables which idealise effects. These usually create partial rate variables,

but not always. *att_var* and *effect_var* are the ELK constructs corresponding to *attvar_fn* and *effvar_fn* in the theory. We have already considered what is required to specify the various kinds of model variables when we described the *_variable* constructs (§ 6.3.2). These latter constructs contain no ecological information. The *_var* constructs, on the other hand specifically relate the ecological attributes and effects to the model variables. We first consider attributes.

6.4.3.1 Attributes

An *att_var* construct is used to define a model variable corresponding to an ecological attribute. To identify such a variable uniquely, we need only name an attribute and an entity. For example, *number* and *wb_pop* can only give rise to a single model variable. To specify the variable fully, we additionally require:

- a name (*e.g.* *n_wb*)
- a range (*e.g.* *real*)
- type of variable (*e.g.* state variable)
- information about obtaining values (*e.g.* initial value = 200)

The system automatically generates a plausible name (*e.g.* *number_wb_pop*) from the attribute and entity names. However, users may prefer to use their own (*e.g.* *n_wb*). The range defaults to the range of the ecological attribute. The type of variable generally defaults to state variable. However, for attributes that are constant (as specified in the *vb* slot in the *att_def* construct), it defaults to parameter. There is also a mechanism for the user to specify their own defaults for how certain attributes of certain types of entities will normally be modelled (see § 6.5.3). The information required about obtaining values depends on the type of variable, as discused in § 6.3.2. Parameters require values, intermediate variables require equation identifiers, etc..

This is similar to the contents of the *_variable* construct. In fact, the *_variable* constructs are not defined explicitly by the user; they are implicitly specified by the *_var* constructs. The *_variable* constructs strip away the ecological information which is not needed to run the model, and fill in a few details making life easier on the user. We give details of this later. The *_variable* constructs differ from the *_var* ones in that for the former:

- a variable type is specified; that information is held in the specific *_variable* construct being used. (*e.g. state_variable*)
- text is given; it is automatically generated by the system.

We illustrate by defining variables that the attributes *number* and *weight* give rise to. For comparison, we repeat the $att\_def_\circ$ specifications in simplified form which create these attributes.

$$att\_var : attribute \times ecol\_ent \times howset \times variable \times \#value \times var\_spec \times value \sqcup eqnid \mapsto bool$$

$$att\_def'_\circ(weight, phys\_obj, positive)$$
$$att\_def'_\circ(number, \#ecol\_indiv, natural)$$

$$att\_var(number, wb\_pop, manual, n\_wb, positive, stvar, ??)$$
$$att\_var(number, predators, manual, n\_pred, positive, parameter, ??)$$
$$att\_var(weight, grs, manual, grs\_wt, positive, parameter, ??)$$

This implicitly specifies one instantiation of the *state_variable* construct, and two of *parameter_variable*. The details of the former are given below, including documentation text.

$$state\_variable(n\_wb, positive, ??,$$
'The attribute *number* of the entity *wb_pop*:#*wb*')

Except for the text which is automatically generated in this case, the state variable that the above *att_var* construct specifies is *exactly the same* as the one created by the *pure_model_var* construct in § 6.3.3. The difference between ecological and pure model variables is not manifest in the implicitly specified *_variable* constructs. Rather, it is in the constructs which were explicitly instantiated by the user (*e.g. pure_model_var* versus *att_var*). This in turn impacts on the ability of the system to make explicit links between the simulation model and the ecological system that it idealises, which is an important aspect of facilitating model comprehension. The same comments apply for effect variables versus pure partial rate variables. These are discussed in § 6.4.3.2.

Note that the value space for *n_wb* has been idealised from natural numbers to positive real numbers. It is very common to model discrete quantities as continuous. This idealisation is explicitly recorded; this would not be possible if *n_wb* was a pure model variable. There would be no link to *number* which has a value space of *natural*.

Note also that we have used one more meta-type: *howset*. This has two instances: *manual* and *auto*. This sort is used both as a directive and as a record keeping device. We shall see that it is possible to have variables created automatically under special user-determined conditions. Here, *manual* means the construct was created manually.

The user may not create net or partial rate variables using the *att_var* construct. The former are only defined implicitly. The latter correspond to a certain class of effect variables which are defined using the $effect\_var$ construct.

6.4.3.2 Effects

Effect variables are defined using the $effect\_var$ construct. To identify an effect variable uniquely, we require exactly what was required to identify a single effect uniquely (as per the arguments to $effect\_fn$).

- an occurrence identifier (*e.g. wb_pred*)
- an attribute (*e.g. number*)
- and the affected role or roles (*e.g.* prey)

The latter can be more than one only in the case where the attribute increases and decreases by the same amount (*i.e.* the system dynamics case). To fully specify an effect variable, we additionally require:

- a name (*e.g. wb_eaten*)
- idealisation choice (*e.g. decrease*)

There is no need to give value space information, as this is obtained from the relevant *att_var* construct. If it is not already specified, a warning is given. What *is* required is to specify which participating entity is being affected and how the effect is to be idealised. Instead of specifying the name of the entity directly, we specify the role and determine the entity from the corresponding *role_esys* specification. This makes it possible to change the participating entity without having to update the effect variable specification.

There are four idealisation options: *increase*, *decrease*, $transfer$, and *change*. The first three options give rise to a single partial rate variable used to increment and/or decrement net rate variables in differential equations. The $transfer$ option corresponds to the system dynamics case when there is a flow from one entity to the other. This means that there is no need to specify separately two effects, one

for increase and one for decrease. Although we are not concerned with biomass in our example model, we have included in the examples below a specification to illustrate the *transfer* option.

The *change* option is useful in two different kinds of situation. One is when addition is not defined on the value space of the affected attribute (*e.g.* for colours). An arbitrary procedure would need to be specified to compute a value for the attribute because incrementing and decrementing makes no sense. It may or may not depend on the value of the attribute in the previous iteration. If it did, we would have something very analogous to a state variable except that we have no differential equation because the value space is not ordered.

The second situation where *change* is useful is when it would be perfectly appropriate to model the effect as a partial rate variable, but the modellers have chosen to do otherwise. The case of the grass growth is a good example of this (see equations 2.2 and 2.10, page 44). In such cases the affect of the $effect\_var$ specification is not to create a new variable, but to give an existing one more ecological meaning. For example the variable *grs_wt* may represent both the weight of the grass, and the process of grass growth.

$$effect\_var : occ \times attribute \times role^{\odot} \times howset \times variable \times chg\_spec \times time \times eqnid \mapsto bool$$

e.g.

$effect\_var(wb\_pred, number, prey, manual, wb\_eaten, decrease, year, eqn\_G)$

$effect\_var(wb\_pred, biomass, [predator, prey], manual, bms\_eaten, transfer, year, eqn\_X)$

$effect\_var(grs\_growth, weight, \_, manual, grs\_wt, change, year, eqn\_A)$

This implicitly specifies two instantiations of the *prate_variable* construct modelling how predation affects *biomass* and *number*. However *no new variable* is created to model how growth affects *weight*. This is because the growth process is not being directly modelled. The *prate_variable* specifications below are exactly the same as the ones that were created manually using *pure_prate_var* (see page 189). Formally:

$prate\_variable(wb\_eaten, positive, eqn\_G, na, n\_wb, Text1)$

$prate\_variable(bms\_eaten, positive, eqn\_X, bms\_predators, bms\_wb\_pop, Text2)$

$grs\_wt = attvar\_fn(grs, weight) = effvar\_fn(grs\_growth, weight, grs)$

*Text*1 is automatically generated as:

> The annual rate of decrease of the quantity represented by the attribute *number* of the entity *wb_pop*:#*wb* due to the process of predation.

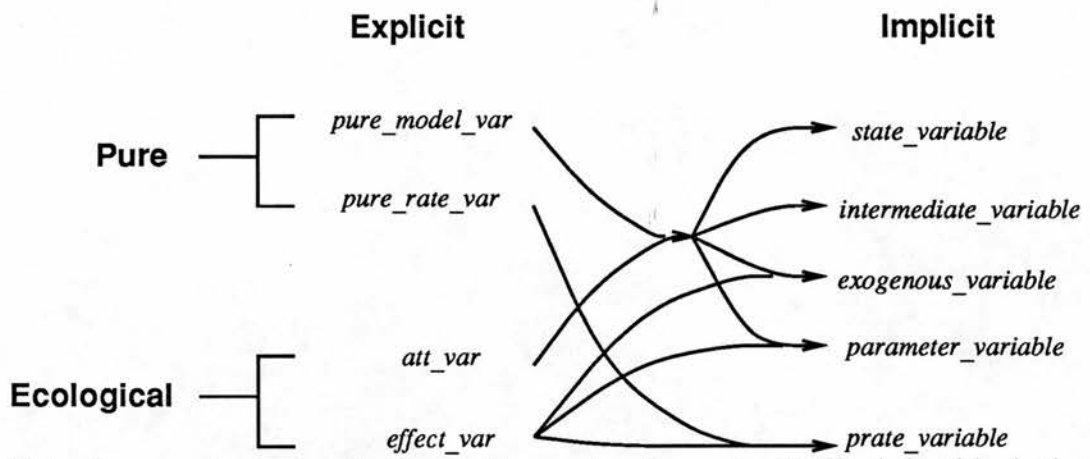
The fourth and fifth arguments in the *prate_variable* specifications are filled in automatically. This works as follows. The *transfer* option says that two state variables are affected, one increases, the other decreases. The *role_esys* construct says what entities participate in which roles; in this case, *predator* is *predators*, and *prey* is *wb_pop* (see page 210). The *effect_def* specification says that the entity in the predator role experiences an increase in biomass. Thus, the variable that goes in the fourth argument is the one corresponding to *biomass* of *predators*. If the variable were not yet defined, (via *att_var*) a warning would be given. Furthermore, it must be a state variable. There is a considerable amount of consistency checking required here; for example although *change* is allowed for growth with respect to *weight*, *decrease* is not. We discuss consistency more in chapter 7.

6.4.3.3 Summary: Model Variables

There are two major classes of variables: pure and ecological. There are two kinds of ecological variables: those that attributes give rise to, and those that effects give rise to. These are explicitly created using *_var* constructs. Ultimately, each variable is implicitly specified with one of five *_variable* constructs. These correspond to the five variable types: state, intermediate, exogenous, parameter and partial rate. Of particular importance are the *att_var* and *effect_var* constructs which in conjunction with *attvar_fn* and *effvar_fn* explicitly link the ecological and runnable-model levels. Figure 6–4 gives a summary. Rate variables are not separately represented; there is one per state variable obtained by applying the function *rate* as we saw in § 6.3.4.

Pure versus Ecological Variables

There are two ways to specify model variables. The preferred way is to associate them with ecological information (via the *att_var* and *effect_var* constructs). The alternative is to define pure model variables using *pure_model_var* or *pure_prate_var*. In either case, the exact same *_variable* constructs are implicitly specified. Only the latter are required for the runnable model. The difference



This diagram shows how the _variable constructs are implicitly defined by both pure and ecological _var specifications. Ultimately, the representation of model variables used to run the simulation contains no ecological information.

Figure 6–4: Model Variables

between pure and ecological variables is chiefly the method used to create the variable; not the variable itself. For ecological variables, ELK provides a range of support facilities to assist in their specification. For pure variables, very little assistance is possible.

One important benefit of using ecological variables is that ELK automatically documents them, users must supply documentation text manually for pure ones. Specifying ecological partial rate variables is easier than specifying pure partial rate variables (*e.g.* ELK automatically infers which state variables get inc/decremented from entity and the attribute corresponding to the effect). Also, there are extensive consistency checking facilities available when specifying ecological variables. However, even for pure variables, ELK provides a certain amount of consistency checking. For example, if a partial rate variable is created, the value spaces for the two state variables that are getting incremented and decremented must be consistent.

As all variables have the type $time \mapsto value$, pure and ecological variables are indistinguishable at the object-level. This is manifest in the use of the *_variable* constructs which does not distinguish between pure and ecological variables. Instead, we make the distinction at the meta-level. A *_variable* specification whose existence derives from a *pure_model_var*, or *pure_prate_var* specification is a pure model variable with no ecological information explicitly associated with it. A *_variable* specification whose existence derives from an att_var, or $effect\_var$ is

an ecological variable. The explicit ecological information is used to generate the documentation text for the appropriate slot in the corresponding *_variable* specification.

What's in a Name?

The atomic variable names (*n_wb*, *wb_eaten*, *grs_wt* etc.) are required only for the users who must not be expected to digest '*_fn*' expressions; *formally, they are unnecessary!* The above *_var* specifications give rise to the following:

$$\begin{aligned} n\_wb &= attvar\_fn(number, wb\_pop) \\ n\_pred &= attparm\_fn(number, predators) \\ wb\_eaten &= effvar\_fn(effect\_fn(wb\_pred, number, wb\_pop)) \\ grs\_wt &= attvar\_fn(weight, grs) = effvar\_fn(grs\_growth) \end{aligned}$$

The '*_fn*' expressions may be viewed as ecologically meaningful names for the model variable. The users never have to see them. In fact, *neither does the system* (except for unpacking expressions with *maximum* as in example on page 177). From an implementation point of view, all the necessary information about variables is included in the '*_def*' and '*_var*' specifications. These are what the system directly uses to give an ecological account of model variables and other idealisation decisions.

Summarising: as far as the *implementation* is concerned, the '*_fn*' expressions are largely superfluous; they are used to give a proper formal account. As far as a *formal representation* of the simulation model is concerned, the atomic names are entirely superfluous; they are convenient for users.

6.4.4 Ecological Schema

The simple schemata described in § 6.3.4 are useful, but do not exploit the rich type structure. There are potentially many equations that might apply in any situation. Some mechanism is required to reduce the number of options. For example, a schema for computing population density should only be offered to compute a variable which represents the attribute *pop_density* of some entity (*e.g. wb_density*). Its inputs are necessarily variables representing the attributes *area* and *number*. Rather than having manually to specify each input, this should be

done automatically. We provide such a facility. Such schemata are called *ecological schema.* An ecological schema contains the following information:

- A name
  e.g. pop_density
- A function output term
  e.g. $pop\_density(\$pop, T)$
- A list of inferrable terms that are required to process the schema
  e.g. $[\$pop]$
- A list of inputs required from the user
  e.g. $[\$region]$
- A procedure call to compute the possible choices for the user inputs
  e.g. $entities\_of\_type\_with\_attribute(area, region, Ans)$
- A list of function input terms
  e.g. $[number(\$pop, T), area(\$region, T)]$
- Precondition(s) which must be true for the schema to be applicable
  $predation(\$predator, \$prey)$
- The function for computing the output
  e.g. $number(\$pop, T)/area(\$region, T)$
- Text describing the purpose of the schema
  e.g. Population density: number per unit area

The \$ denotes dummy variables which are instantiated when the schema is activated. In the example model, the *pop_density* schema is used twice. $\$pop$ is either *wb_pop*, or *alt_prey*; $\$region$ is *serengeti* in both cases. The function input and output terms are exactly analogous to the inputs and outputs in pure schema.

Unlike the pure schemata, the user need not specify the function inputs directly. For example, if the user wishes to specify how to compute the variable *wb_density*, ELK knows that this is the variable corresponding to the attribute *pop_density* of the entity *wb_pop*. Thus, $\$pop$ is known to be *wb_pop*. This determines one of the function inputs. The other one requires knowing what the region is that the population ranges over. Using the procedure call in the schema, ELK finds the list of entities of type *region*. This list of entities is ordered so that those for which model variables have been created corresponding to *area* come first (*e.g. serengeti*, but not *plains*). Priority is also given to those which the user has noted interest in with respect to the attribute *area* (noting interest is covered in

§ 6.5.2). This helps ensure that the most likely choices are first. In this case, the user would choose *serengeti*. This is substituted for *$region*, thus determining the other function input. ELK then automatically creates the required *ischema*, *inarc*, *outarc*, and *tie* specifications. The end result is exactly as it would be if a pure schema was used (analogously, the end result of specifying pure and ecological model variables is the same). ELK provides much much more assistance when ecological schemata are specified. In particular:

- The number of available schemata to choose from is dramatically reduced
- Less work is required to specify the function inputs.

In general, the user inputs will be fewer and easier to deal with than the function inputs. They are fewer, because often the system can infer them directly from the output variable. They are easier to deal with because ELK uses type information to compute the possible choices for these inputs. This is evident from this very simple schema. For more complex schemata, the savings are much more dramatic (see example in § 7.5.2.2).

6.5 Dialogue Level

We shall consider three kinds of construct at the dialogue level:

- goals
- interest/importance
- user-specified defaults

6.5.1 Goals

Goals were discussed at length in chapter 3. We concluded that virtually all goals ultimately are cast in terms of attributes, effects, and/or their corresponding model variables. We referred to these as X and Y in the specific goal types. The following are actual stated goals regarding the modelling exercise described in [Hilborn & Sinclair, 1984].

- How big will the wildebeest population get?
- What is the affect of the increased size of the wildebeest population on the [size of the] aggregate population of alternate prey of the lions and hyena?

The first can be readily viewed as finding the value of an attribute, a specialisation of the goal type: "To plot X versus Y". This corresponds to characterising what an output of the model should be. In this case, Y is some time or time period. The attribute is 'maximum number over some period of time'; it applies to the wildebeest population. The time period would probably be some chosen number of years.

The second is readily seen to be of the type: "What is the affect of X on Y?". Here X is the wildebeest population, C_x (the change in X) is 'increase' and Y is the aggregate population entity. For now we take these to be questions relating to the ecological system, rather than to the model. Thus, we use the constructs from the ecological level to represent these goals. For now, assume that X and Y are unary functions from time periods to some kind of numbers. Formally, we represent these goals as follows:

$$output(\lambda P{:}set(time).qnam\_tim(maximum, number)(wb\_pop, P), 80s)$$

$$xy\_affects(\lambda T{:}time.number(wb\_pop, T), increase, \lambda T{:}time.number(alt\_prey, T))$$

$$\begin{array}{lll} \forall T \sqsubset^s time. \; V \sqsubset^s value. & & \\ \quad output: & (T \mapsto V) \times time & \mapsto bool \\ \quad xy\_plot: & (T \mapsto V) \times (T \mapsto V) & \mapsto bool \\ \quad xy\_affects: & (T \mapsto V) \times (T \mapsto V) & \mapsto bool \end{array}$$

We have intentionally used the full types rather than using the meta-types *attribute*, $effect$, or *variable*. Goals such as in the above examples may be interpreted either at the ecological or runnable-model level. We have formalised them using ecological information. If we instead used the corresponding model variables, the goals become relevant to the structure of the model, what outputs it should have, and/or what experiments can or should be run on the simulation model. Specifically, the first goal would say that an output of the model should be the value of the variable representing maximum number of wildebeest in the simulated ecological system evaluated for some particular time period. In the simple Serengeti model, this is constant for a single simulation run, but could vary for different runs if different parameter settings were used for some experiment; alternatively different time periods might be of interest.

If cast in modelling terms, the second goal suggests that there needs to be some mechanism for causing the model variable corresponding to the number of wildebeest to increase so that the effects of this can be monitored. This has

implications on the kind of variable it can be and the specific experiments that need to be run on the model. For instance, if the variable is a state variable it will not be easy to run controlled experiments to measure the affect of increased values of the variable because the values of a state variable are not directly controllable. Rather, they are subject to potentially many changes due to various processes.

6.5.1.1 Other Goal Concepts

In § 3.4 we noted that C_x could be different equations for computing a model variable, or to include or exclude something from the model. We do not give any details here, but note that the design of ELK naturally accommodates the use of such notions. For instance, in § 6.5.2 we describe a mechanism which will allow aspects of the model to be selectively ignored. There are specific constructs for specifying what equations are used to compute model variables. Simulation experiments could readily be performed by running the simulation using different equations.

In § 3.4 we noted that concepts like *equilibrium* and *threshold* are analogous to the higher-order functions *average*, *maximum*, etc. Formally we represent *equilibrium* in the following way. Equilibrium takes a unary function from times to some kind of numbers, and a set of times over which the function applies and returns a number. For example, consider the state variable *n_wb* representing the simulated number of wildebeest. The computation of the equilibrium value is realised by running the model for some length of time. This length of time is specified as a set of times, for example 24 hrs a day for 5 days. Usually, it will be whatever time is specified for the model overall. The type of *equilibrium* and an example is given below.

$$\forall V \sqsubseteq value.\ equilibrium : (times \mapsto V) \times set(times) \ \mapsto \ real$$

$$equilibrium(\lambda T.n\_wb(T), 5days) = 20$$

The example says that the equilibrium value of *n_wb* as measured by a running of the model over a particular 5 day interval ($5days{:}set(time)$) is 20. The unary function in this case is of type $times \mapsto positive$, which is ok since $positive \sqsubset real$.

The concept of a threshold is a bit more complex. Let us use examples to clarify. In [Legovic, 1987] there was a hypothesis that if the minimum annual

sea temperature was below a certain threshold this would cause a catastrophic decline in the population size of the jellyfish the following year. Another example is the threshold level of predation above which the wildebeest population begins to decline. In both cases, a threshold is defined explicitly in terms of two quantities. Broadly speaking, a threshold value of $V1$ with respect to $V2$ means that there is some value $V1_t$, such that for some 'significant' range of values of $V1$ less(more) than $V1_t$, the value of $V2$ does not change 'significantly'. But when $V1$ is equal to or more(less) than $V1_t$, then the value for $V2$ does change 'significantly'. Note that a threshold only makes sense on totally ordered value spaces. The notion of 'significance' would need to be formalised for each application.

The type of *threshold* is similar to that of *equilibrium*. It applies to a unary function on times to reals, and a set of times. However, we need an additional argument to refer to the other variable. In modelling terms, the unary function may be a state variable or an external variable.

$$\forall T \sqsubset^s time. \forall T_{n1}, T_{n2} \sqsubset^s real$$

$$threshold : (T \mapsto T_{n1}) \times (T \mapsto T_{n2}) \times set(T) \mapsto real$$

$$\begin{aligned} threshold(\quad & \lambda Y : years.annual\_min\_sea\_temp(Y), \\ & \lambda Y : years.n\_jelly(Y), \\ & \{1980, 1981, \ldots, 1989\} \qquad) \end{aligned}$$

The expression in the example returns threshold value (if it exists) of annual minimum sea temperature with respect to the number of jellyfish for the 10 year period 1980-1989. Note that this representation assumes that the annual minimum is a simple value not computed explicitly for different times during the year. Alternatively, we could make this explicit. For example, the annual minimum might be computed from 12 monthly values.

$$\begin{aligned} threshold(\quad & \lambda Yr : set(month).minimum(\lambda M : month.sea\_temp(M), Yr), \\ & \lambda Y : year.n\_jelly(Y), \\ & \{1980, 1981, \ldots, 1989\} \qquad) \end{aligned}$$

6.5.2 Interest

There are many things that a modeller might be interested in, varying from quite general to rather specific. We provide a class of constructs which enable such

interest to be explicitly noted. We refer to them as the *_interest* constructs. They serve three primary purposes:

1. To provide hooks into the specification process. Whatever the user expresses interest in, the system comes back with a variety of suggestions about how that might be pursued and elaborated on.
2. To record explicit modelling decisions; a user can express interest in something, (thus recording its importance) but then consciously decide to ignore it for a particular simulation model.
3. To facilitate experimenting with slight variations of a model (related to previous point).

The first purpose is enabled simply by having the interest constructs. To accomplish the second purpose, we provide a special slot in each *_interest* construct whose value is one of *heed* and *ignore*. Some examples with their formal representations are listed below.

- a type of entity; *e.g.* animal populations
  $type\_interest(set(animal), manual, heed)$
- a specific entity; *e.g.* the wildebeest population
  $ent\_interest(wb\_pop, manual, heed)$
- attribute of a sort of entity; *e.g.* the size (in numbers) of animal populations
  $att\_type\_interest(number, set(animal), manual, heed)$
- attribute of a specific entity; *e.g.* the size (in numbers) of the wildebeest population
  $att\_ent\_interest(number, wb\_pop, manual, ignore)$
- a type of process; *e.g.* predation
  $proc\_interest(predation, manual, heed)$
- the effect a process has on some attribute *e.g.* predation affects *number*
  $effect\_interest(predation, biomass, manual, ignore)$
- an occurrence of a process; *e.g.* lions preying on wildebeest
  $occ\_interest(ln\_wb\_pred, manual, ignore)$
- an effect of a process occurrence *e.g.* numbers of wildebeest
  $effect\_occ\_interest(wb\_pred, number, wb\_pop, manual, heed)$

Until now, we have given the impression that the *att_var* and $effect\_var$ constructs necessarily create model variables. While it is true that to define ecological

model variables you have to use these constructs, they may be overridden. When the *ignore* option is used for the *att_ent_interest* or *effect_interest*, this causes the variable specified using *att_var* or *effect_var* to *not* be created.[4] If *ignore* has been specified *before* a *_var* construct has been instantiated then this is a cue for the system to not bother asking the user about it.[5] If is specified *after*, then the *_var* specification stays around in case it is needed again later, possibly to perform experiments with the model structure itself.

There are a variety of additional useful things that may be done with respect to these constructs, none yet implemented. The suggestions could directly link with an agenda mechanism which executed commands automatically when users selected them. The ignore option would remove the item from the agenda. These interest specifications could be automatically created by the system when goals (such as in the above examples) are specified. The penultimate argument of these constructs records whether the instantiation of an interest construct has been manually or automatically specified. Currently, it is always set to *manual*. The ignore facility for the *att_ent_interest* construct is fully operational.

manual here means the same thing as it did for the *_var* constructs, that the construct was created manually. Because goals necessarily imply interest in the things they mention, interest constructs may be automatically specified.[6] We introduce another meta-type for the heed/ignore slot. These meta-level constructs are typed as follows:

---

[4] The process related *_interest* specifications are not yet implemented.

[5] Not implemented yet.

[6] Not implemented yet.

$ig\_spec \sqsubset^s spec$	$ignore{:}ig\_spec;\ \ heed{:}ig\_spec$		
$sort\_interest:$	$\mathcal{P}(ecol\_ent) \times howset \times ig\_spec$	$\mapsto$	$bool$
$ent\_interest:$	$ecol\_ent \times howset \times ig\_spec$	$\mapsto$	$bool$
$att\_type\_interest:$	$attribute \times \mathcal{P}(ecol\_ent) \times howset \times ig\_spec$	$\mapsto$	$bool$
$att\_ent\_interest:$	$attribute \times entity \times howset \times ig\_spec$	$\mapsto$	$bool$
$proc\_interest:$	$process \times howset \times ig\_spec$	$\mapsto$	$bool$
$effect\_interest:$	$process \times attribute \times howset \times ig\_spec$	$\mapsto$	$bool$
$occ\_interest:$	$occ \times howset \times ig\_spec$	$\mapsto$	$bool$
$effect\_occ\_interest:$	$occ \times attribute \times howset \times ig\_spec$	$\mapsto$	$bool$

6.5.3 User Specified Defaults

There is a class of constructs whose instantiations act as default rules for idealising specific kinds of things in the description of the ecological system. The following specification is a rule which says that every ecological model variable that corresponds to the attribute 'capture coefficient' of an animal population is represented as a parameter by default. It also indicates the default value space[7].

$$att\_inh'(cap\_cf, set(animal), positive, parameter)$$

Other useful defaults may be set. This includes

- a dimension value space,
- a default value
- and whether variables should be automatically created corresponding to that attribute each time an instance of that type is created[8]. This is the second use of the *howset* meta-type: as a directive, not merely as a record. Its instances are *auto* and *manual* with the obvious meanings.

The complete version is typed:

$$att\_inh : attribute \times \mathcal{P}(ecol\_ent) \times (\#value)^2 \times var\_spec \times value \times howset \mapsto bool \tag{6.1}$$

There is an analogous $effect\_inh$ construct which is used to specify how effects are idealised by default. For attributes and effects, when no explicit default rule is

---

[7] We use the suffix '*_inh*' because defaults are inherited from these specifications.

[8] Partially implemented.

given, the system obtains the defaults from the corresponding *_def* specification. These are also used by the system to load the default defaults when a user is instantiating an *_inh* construct.

This is a good example of our general design philosophy to alleviate the users becoming overwhelmed by a bewildering number of possibilities. There are always plenty of defaults slipping through from general/ecological knowledge base. Ecologists can use a small subset of the available features and remain unaware of the real power of the system until such time as they require it. Even for experienced users, the idea is for the extra power never to intrude, but only to help when needed. The extent to which we have succeeded in this will not be known for sure until more empirical testing is done.

Logical Implications

Instantiations of these three constructs are not directly part of the object-level description of the ecological system. Nor do they define any object-level primitives, and so are not meta-level constructs in the usual sense. Their role is to assist in the elicitation of meta- and object-level specifications. As such they are expressed in logically precise terms which are slotted into the meta-level specifications. In this sense, they may be viewed also as meta-level constructs, but to avoid the confusion of having two kinds of meta-level constructs, we simply call them dialogue level constructs.

6.6 Implicit Specification

Implicit specification is a major theme in the theory and implementation of Elk-Logic. Compared to the total number of instantiated constructs, relatively few are explicitly asserted in the Prolog database. Even fewer are directly specified by the user. There is a tradeoff between computational speed, and the economy and modifiability derived from implicit specification. Occasionally, we have found it necessary to assert inferred specifications explicitly. Examples we have seen of this include keeping track of the valid sorts and of the list of value instances for qualitative value sorts. Sorts are inferred from the sort hierarchy, but explicitly asserted using the *sort*/1 relation. The entity types, on the other hand, are implic-

itly specified in terms of the sorts. Here we summarise some of the main examples of implicit specification in ELK.

- the entity type hierarchy is induced from the sort hierarchy using *set*, $\sqcup$, and $\setminus_t$.
- transitive closure (e.g. see definition 5.2, page 134)
  $\sqsubset^s_\circ$ & $\sqsubset^s$, $\prec^p_\circ$ & $\prec^p$, $\subset_\circ$ & $\subset$, $\subset^p_\circ$ & $\subset^p$
- possible component relation ($\subset^p$)is induced from $\#$, $\sqsubset^s_\circ$, and $\prec^p_\circ$.
- attributes: *att_def* induced from att_def_b, att_def_m, $\sqsubset^s$, *set*, *qnam_ent*, and *qnam_tim*.
- *_variable* constructs induced from *_var* constructs
- differential equations implicit in *_variable* constructs.

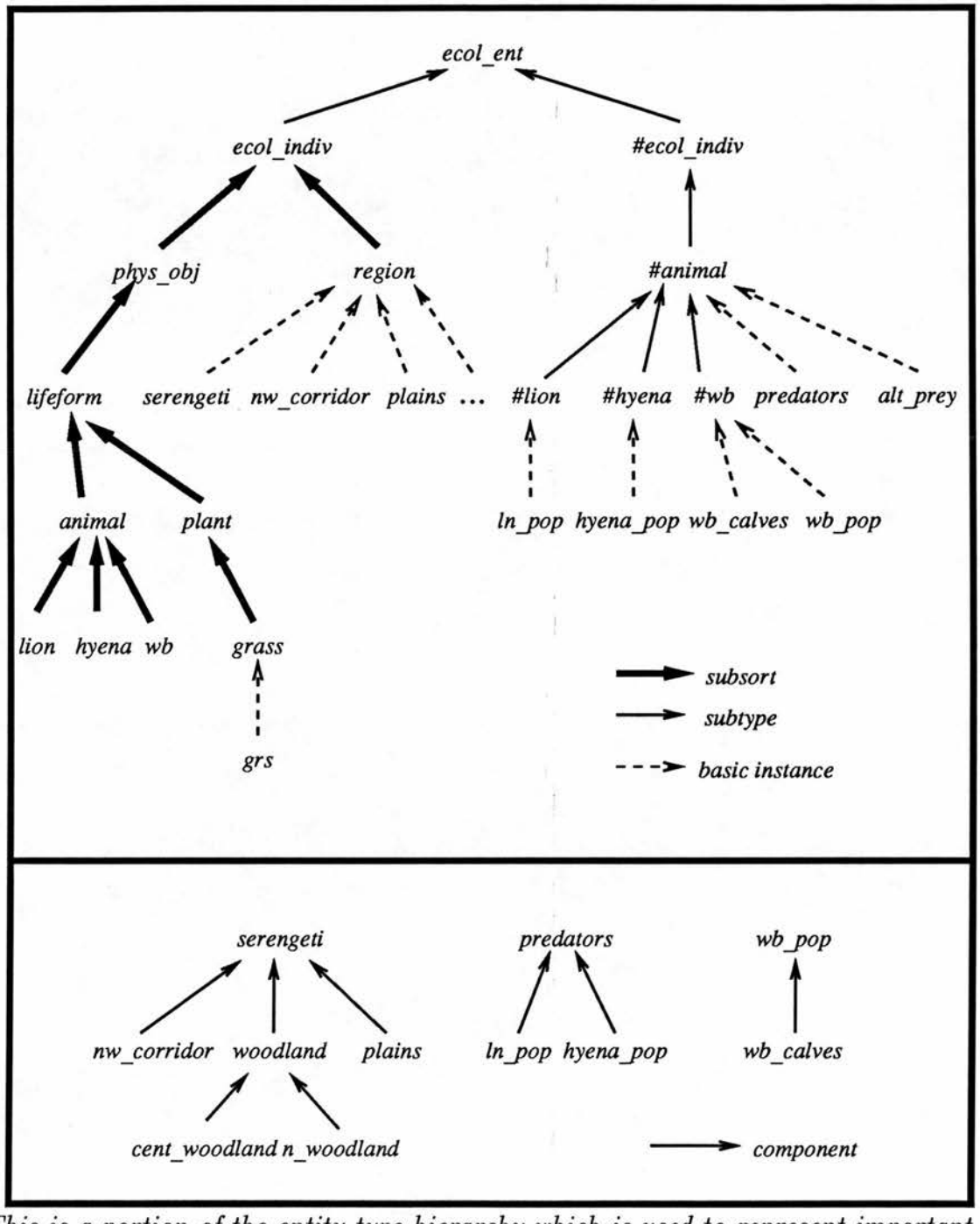
6.7 The Serengeti Revisited

Except for low-level details, we have now fully described the theory and implementation of the formalism used by ELK. With it, we can represent ecological knowledge, systems, and models, as well as additional information useful for controlling the elicitation process (*i.e.* the dialogue level). Rather than replace the logic representation of the simple model presented in chapter 2, we have augmented it. We now give a summary of the use of this augmented representation for the simple Serengeti model. It is important to note that we present but one of a large number of possible ways to represent the required information. The particular details would vary according to the needs and tastes of different users. We shall consider various alternatives and extensions and their tradeoffs in the next section.

6.7.1 Types, Entities, Substructure, Attributes

A summary of the entities, types, and substructure is given in figure 6–5. We have discussed most of the details already. Next we consider attributes.

Physical objects have the attribute *weight*; sets of any entity have the attribute *number*. Wildebeest, grass, animals and plants etc. inherit *weight* from physical objects. Any set entity inherits *number* from the most general type of set entity. The value space for *weight* is positive real numbers, and for *number* is natural numbers. Other attributes that we are concerned with include area, rainfall, capture



This is a portion of the entity type hierarchy which is used to represent important ecological information for the simple Serengeti model.

Figure 6–5: Type Hierarchy, Entities and Substructure

coefficient, fecundity, specific rate of survival, and handling time. In this context, handling time is the amount of time it takes for a predator to deal with its prey.

These latter attributes are used to describe average characteristics of animal populations. For example, the parameter *wb_htime* represents the average value of the attribute handling time for all the individual wildebeest in *wb_pop*. We represent 'average handling time' in terms of 'handling time' using the higher order function, *average*. Although it is often convenient to speak of a fictitious 'average member' of a group, we do not represent such entities explicitly. It is sufficient to reason about average values which are properties of the whole set.

We proceed as follows: we use *att_def* to create the attribute *htime* defined on animals. This induces the attribute $qnam\_ent(average, htime)$ which applies to sets of animals. For example, $qnam\_ent(average, htime)(alt\_prey, T)$ denotes the average handling time (per animal) of the alternate prey population at time T. Note that there is no need to perform any computation using *average*, if as is often the case, there are no members specified. It may be treated exactly as any other attributes. If individuals were later added, the computation could then be performed as required.

While this attribute could in principle vary, it almost never will for a particular simulation model. Thus, we create an *att_inh* specification to indicate that every time a variable corresponding to the average handling time of a set of animals is created (using *att_var*) the default for the slot specifying the type of variable will be *parameter*. If, as in the case for fecundity of wildebeest, there is a value which holds for all wildebeest everywhere, then this may rightly go in the general/ecological knowledge base in an *att_def* specification. This is not to say that it should and always will be constant for every wildebeest population, only that the default value is universally applicable. An *att_inh* specification should be used if it is known that it will mostly be used as a parameter and/or if the default value is for some reason different in the particular ecological system of interest (say 4). In the *_inh* specifications, *manual* means that variables are manually created for every entity of the specified type. Where *na* occurs in the specifications below, this refers either to the dimension value space or default value slots. Formally:

$att\_def(htime, animal, positive, na, variable, na)$
$att\_def(qnam\_ent(average, htime), \#animal, positive, na, variable, na)$
$att\_inh(qnam\_ent(average, htime), \#animal, positive, na, parameter, na, manual)$
$att\_var(qnam\_ent(average, htime), wb\_pop, manual, wb\_htime, positive, parameter, 30)$

$att\_def(fecundity, animal, positive, na, variable, na)$
$att\_def(qnam\_ent(average, fecundity), \#wb, positive, na, variable, .5)$
$att\_inh(qnam\_ent(average, fecundity), \#wb, positive, na, parameter, .4, manual)$

We also use this technique for capture coefficient and specific rate of survival. For 'specific rate of survival', there is a slight variation. Attributes of populations which are expressed as 'specific A' where A is some attribute (*e.g.* rate of survival) mean *per individual.* There would be little point in referring to the specific rate of a single individual. Thus the approach we used for *htime* etc will not work in exactly the same way. Instead we do the following: we have a single attribute *r_surv* which applies to individuals of sort *animal.* This induces *qnam_ent(average,r_surv)* whose precise meaning is: "specific rate of survival". Appropriately, this applies only to sets of animals, not individuals. The absolute rate of survival of the population is the total rate which is represented as: *qnam_ent(total,r_surv).*

This illustrates the general principle of using relatively few primitives in conjunction with various combining functions to avoid proliferation of names and achieve greater expressive power. We can use our primitives to incorporate the notion of specific attributes (with respect to populations) directly in the representation. We also can achieve the effect of representing and reasoning about the important concept of average individuals in a principled uniform way.

Recall that the specific rate of survival of the wildebeest is different for the adults and the calves. Recall also that, as explained in chapter 2, the wildebeest calves have no explicit existence in the simulation model. This raises the question of how and whether to represent the calves. Unless a user is content to represent the specific rate of calf survival as a pure model variable and document it manually, we must distinguish between the two age groups for the wildebeest population. There are various ways to do this. To get the most from the system, we should subdivide the wildebeest population by age into the two sub-populations. However, for illustrative purposes, we instead use the simpler approach and just create the entity $wb\_calves : \#wb$. For this simplicity, we tradeoff the system 'knowing' about the age and the substructure which could be used for automatic model

documentation. Also, if the model were later expanded to include a more detailed representation of the calves, it would be easier to do this if the substructure was already defined.

There are problems with trying to play the same game with the specific rate of predation of wildebeest by the predator population. Suppose we created an attribute called *r_pred* representing rate of predation. The problem is that the attribute $qnam\_ent(average, r\_pred)$ which represents *a* specific rate is ambiguous. In general, there would be a different such specific rate for each different predator and prey combination. To deal with this, we use a higher-order function to represent a family of attributes representing specific rates of predation for various predators. In particular the specific rate of predation of wildebeest by the predator populations is represented as $r\_pred(predators)(wb\_pop)$, etc. Formally:

$$r\_pred : \#animal \mapsto (\#animal \times time \mapsto positive)$$

For illustrative purposes, we have one pure model variable for this model: cf_born. It is an intermediate variable which refers to the number of calves born each year. A hypothetical user might reason that it is quicker and simpler initially to use a pure model variable for this because wildebeest reproduction is modelled in a way that does not distinguish calves from adults. However in general, it would be a better idea to make an explicit ecological connection as follows. The attribute 'number of young born each year' which applies to animal populations is naturally represented as the total fecundity. Formally:

$$qnam\_ent(total, fecundity)$$

The complete list of entities and their attributes which give rise to model variables, as well as the pure model variables is given in table 6–1.

6.7.2 Processes

There are several processes relevant to this modelling exercise. These include growth, reproduction, mortality, precipitation, and predation. We have considered predation already. It occurs between animal populations. One necessary effect of predation is to decrease the *number* attribute of the prey population. A possible effect is to increase and decrease the amount of DDT in the predator and prey populations, respectively; there may be no DDT around.

Entities	*Variable*	*Attributes*	*Processes*
wb_pop	*n_wb*	*number*	*wb_pred*; *wb_repro*
			adult_mort
	wb_density	*pop_density*	
	spr_wb_surv	$qnam\_ent(average, r\_surv)$	
	wb_cap_cf	$qnam\_ent(average, cap\_cf)$	
	wb_htime	$qnam\_ent(average, htime)$	
	wb_fec	$qnam\_ent(average, fecundity)$	
	spr_pred_wb	$spr\_pred(predators)$	
serengeti	*dry_ssn_rain*	*rainfall*	
	area_sgti	*area*	
wb_calves	*spr_cf_surv*	$qnam\_ent(average, r\_surv)$	
predators	*n_pred*	*number*	*wb_pred*; *ap_pred*
alt_prey	*ap_density*	*pop_density*	*ap_pred*
	ap_cap_cf	$qnam\_ent(average, cap\_cf)$	
	ap_htime	$qnam\_ent(average, htime)$	
grs	*grs_wt*	*weight*	*grs_growth*
ln_pop			
hyena_pop			
	cf_born	pure model variable	

This is a summary of the important entities, their attributes and the processes which affect them. We only mention the attributes and processes that ultimately give rise to model variables. We include the lion and hyena populations here because they compose the aggregate predator population.

Table 6–1: Serengeti: Entities, Attributes, Variables and Processes

Table 6–2 summarises the contents of the general/ecological knowledge base with respect to processes. It is a tabular presentation of the corresponding *role_def* and *effect_def* specifications. Similarly, table 6–3 summarises the key information about processes at the ecological system level. These are tabular versions of the *role_esys* and *effect_esys* specifications. Although there are five process occurrences, there are only four effect variables. The occurrence *ap_pred* is ignored. In three of the five cases, there are no interesting idealisation decisions that have been made. The effects are modelled just as in the real system. For grass growth, rather than model it as a state variable and incrementing it each iteration, it is computed directly from scratch each iteration. This idealisation choice is explicitly recorded in the corresponding *effect_var* specification as *change* instead of *increase* which is the default taken from the *effect_def* specification. A more drastic idealisation has been made with respect to the occurrence *ap_pred*, which is not represented in the simulation model at all! Nevertheless, it plays an important role in the specification of the model (details in § 7.5.2.2). The full details of the specification of ecological information for the simple Serengeti model are found in appendix D.

PROCESS	*ROLES*		*EFFECTS*		
precipitation	*agent*	*ecol_ent*	*agent*	*rainfall*	*increase*
mortality	*agent*	*lifeforms*	*agent*	*number*	*decrease*
			agent	*biomass*	*decrease*
growth	*agent*	*lifeforms*	*agent*	*weight*	*increase*
			agent	*height*	*increase*
reproduction	*agent*	*lifeforms*	*agent*	*number*	*increase*
predation	*predator*	*#animal*	*predator*	*number*	*increase*
	prey	*#animal*	*[predator, prey]*	*biomass*	*transfer*

*This table summarises the contents of the general/ecological knowledge base regarding processes. We have included some extra information that is not required for the Serengeti example (*e.g. *precipitation, height).*

Table 6–2: Processes: General/Ecological Level

6.7.3 Alternatives and Extensions

So far, we have described a minimal general/ecological knowledge base and corresponding description of the Serengeti ecological system. We have provided little more than the barest essentials. In any realistic use of the system for modelling the

PROCESS	*OCCURRENCE*	*ROLES*		*EFFECTS*		
mortality	*adult_mort*	*agent*	*wb_pop*	*agent*	*number*	*decrease*
growth	*grs_grwth*	*agent*	*grs*	*agent*	*weight*	*increase*
reproduction	*wb_repro*	*agent*	*wb_pop*	*agent*	*number*	*increase*
predation	*wb_pred*	*predator*	*predators*	*prey*	*number*	*decrease*
		prey	*wb_pop*			
predation	*ap_pred*	*predator*	*predators*			
		prey	*alt_prey*			

This table summarises the description of the ecological system regarding processes. Except for the predation of the alternate prey, we only list the occurrences and corresponding effects that give rise to model variables. Occurrence 'ap_pred' is required in order for the equation for computing the specific rate of predation of the wildebeest population to be applicable (equation 2.7, chapter 2). Note that these are the 'real' effects, not the variables which idealise them.

Table 6–3: Processes: Ecological System Level

Serengeti, it is likely that there would be much more ecological information provided, both general ecological knowledge (relevant to ecosystems like the Serengeti) and specific information about the Serengeti itself.

In this section, we consider various ways that the ecological information for the same model might have been represented differently, or more extensively. We also briefly consider how the model itself could be extended in a number of ways to include more details and additional factors. Our motivation for describing these alternatives and extensions is twofold. First, it enables us to demonstrate a wider range of expressive capability of our ecological modelling formalism. A major design feature of the representation is to achieve generality and economy through reuse of primitive concepts. Second, we illustrate the benefits of gaining this expressive power, and how our general representation techniques facilitate these. The main benefits of having a more elaborate description of the ecological information are:

- to ensure greater model comprehension (even if the model stays the same)
- to facilitate extending the model in a simple natural way
- to facilitate performing experiments with model structure.

The important parts of the ecological system that are described but not modelled serve to record modelling decisions that were made (*i.e.* idealisations, sim-

plifications etc.). However, having the extra information also makes it easier to extend the model if desired.

Any given aspect or concept relating to the model, the ecological system, or general/ecological knowledge can be represented in a variety of ways differing in richness and generality. If a concept is likely to be used only once, or if it has no obvious direct ecological interpretation, then there is no need to represent it at all except in the equations describing the model. On the other hand, if a concept is likely to be used many times in many different contexts, then there are advantages in representing the concept in a richer more general way.

We add the following new subsorts of *animals*: *zebra*, *warthogs*, *kongoni*, *topi*, *impala*, and *gazelles*. We also create populations of each of these (*e.g.* *zeb_pop*: $\#zebra$, *tp_pop* : $\#topi$). We define the substructure of the *alt_prey* aggregate population just as we did for the *predators* (*e.g.* *topi_pop* $\subset_o$ *alt_prey*). The part hierarchy does not need updating for this. It is permitted by the *d_subdiv* case in the definition of the $\subset_o^p$ relation (see (5.29), chapter 5).

There are three main regions, the woodlands, the plains, and the western corridor. The woodlands are further subdivided into the central and northern sections. We create new the region entities *woodland*, *plains*, *nw_corridor*, *n_woodland*, and *cent_woodland*. We define substructure as follows: *plains* $\subset_o$ *serengeti*, *woodland* $\subset_o$ *serengeti*, *w_corridor* $\subset_o$ *serengeti*, *cent_woodland* $\subset_o$ *woodland*, and *n_woodland* $\subset_o$ *woodland*. Since $\subset$ is the transitive closure of $\subset_o$, we can easily infer that *cent_woodland* $\subset$ *serengeti*, etc. This defines a simple regional hierarchy for the serengeti which serves as a part of the description of the ecological system being modelled (see figure 6–5). None of the regions are in fact used for anything in the simple model. However, more complex models could make use of them. To permit this substructure, we require *region* $\prec_o^p$ *region*. This signifies that *region* is homogeneous.

We add the attribute *location* which has *places* as its value space in the corresponding *att_def* specification. We can view a region as a kind of a place, so we insert *place* between *ecol_indiv* and *region* in the sort hierarchy. This sets up some of the required machinery to represent the process of migration, if necessary. We also add the attribute *prot_content* with value space $[0, 100] : set(reals)$. The entity *grs* is subdivided according to the protein content. We define two instances of *grass*:

$$att\_dim\_fn_1(grs, prot\_content)(food)$$
$$att\_dim\_fn_1(grs, prot\_content)(roughage)$$

where $\{food, roughage\}$ is the dimension value space used. We place this in the appropriate slot in an *att_inh* specification for the attribute *prot_content* of *grass* entities. We record the meaning of these categories using the *interval* construct (*e.g.* $roughage = [0, 7]$ and $food = [8, 100]$). Similarly, we subdivide the wildebeest population *wb_pop* by age into $dim\_fn(wb\_pop, age)(calf)$ and $dim\_fn(wb\_pop, age)(adult)$ (note that the former denotes exactly the same thing as *wb_calves* in the ecological system, but contains more information).

It is useful to consider the processes of growth and precipitation. Both are modelled in the same way, however for illustrative purposes, we chose to have an explicit representation for growth, but not for precipitation. The process of precipitation which produces rainfall during the dry season is very important in the Serengeti. It is specifically taken into account in the simple model, albeit in a rather simplistic manner. The amount of rainfall during the dry season is a parameter of the model, and thus constant for any simulation run. It is used to compute the amount of grass that is available. Each year, grass grows a certain amount which depends on the amount of rainfall during the dry season (see equation 2.2). Someone constructing that simple model would be free to decide which of the growth and precipitation processes to represent explicitly in the general/ecological knowledge base and/or in the description of the ecological system. They might well choose neither. Making the representations explicit is more work, and to some extent complicates the model description. For example, for the purpose of the model itself, the existence of the entity *grs* and of the attribute *weight* is entirely superfluous. However, there are benefits derived from having the extra ecological information. These may or may not be considered worth the effort; it depends largely on the frequency and diversity of the use of these concepts. This is for the user to decide.

This concludes the treatment of the expressive power issue. We now consider how the formalism we have described facilitates achieving model comprehension and reducing conceptual distance.

6.8 Model Comprehension

During and especially after a model is specified using ELK, a plethora of information is available that can be used to explain the simulation model in terms of the ecological system that it represents. This assumes that a user has made the effort to create mostly ecological variables rather than pure ones. The key technique which facilitates model comprehension is the separation of the two ecological levels from the simulation modelling level. The understanding of the model that the system has derives entirely from its ability to describe the simulation model in ecological terms, and vice versa. The major facilities that exist in this respect are:

- *automatic documentation:* of each ecological model variable.
- *explicit idealisation:*
  - the differences between the *_def* specifications and the corresponding *_var* specifications.
  - something in which interest is explicitly noted, but is explicitly excluded from the simulation model using the *ignore* option.
- *implicit idealisation:* something is included in the description of the ecological system, but is not included in the simulation model.

Automatic Documentation

As an example of automatic documentation, the system generates the following text to explain the meaning of the variables *n_wb* and *wb_eaten*.

> "*n_wb*: the attribute *number* of the entity *wb_pop* : #*wb*"
>
> "*wb_eaten*: the annual rate of decrease of the quantity represented by the attribute *number* of the entity *wb_pop* : #*wb* due to the process of predation"

This alleviates the need for manual program documentation as in figure 2–2.

Documentation for subdivisions created using attribute-based substructure is also provided:

```
roughage_grs: A grass which has the following attributes:
     - prot_content is in the interval: 0-7
       which is referred to as "roughage"
```

```
calf_wb_pop: The subdivision of wb_pop:#wb such than for
             each member:
      - age is in the interval: 0-1
        which is referred to as "calf"
```

The user only sees the abbreviated names generated by ELK, not the $att\_dim\_fn_i$ terms (note that *calf_wb_pop* is ELK's name for what we have been calling *wb_calves*). This is verbatim text generated by ELK. Note that depending on whether the substructure is defined on an individual or a collection, the relationship between the attributes and the entities differs; ELK generates appropriate text accordingly. The grammatically incorrect "a grass" arises because ELK does not currently make a distinction between mass nouns and count nouns. This point is raised in chapter 8. For the variables, this is the text that goes in the corresponding [implicit] *prate_variable* specification. Slightly different text is generated if the $transfer$ option is used in the $effect\_var$ specification. We have "...rate of transfer of stuff represented ..." instead of "...rate of decrease of the quantity represented ..." to reflect that it is modelled as a flow of some material. More verbose versions would be easy to provide where, for example "the entity *wb_pop* : $\#wb$" is replaced instead by "the collection of wildebeest, *wb_pop*". It would be equally easy to have have text descriptions as part of the general/ecological knowledge base which can be appealed to for detailed explanations of the meaning of each entity, attribute, and/or process. Attributes like weight are fairly self-explanatory, however something like handling time might require a phrase or two to document satisfactorily.

Explicit Idealisation

Variables specified using att_var constructs are idealisations of attributes specified using att_def constructs. Differences between the corresponding slots in these constructs correspond to idealisations. For instance, the number of wildebeest is in reality measured only in natural numbers, but is idealised on positive reals (*n_wb*). The number of predators is in reality variable, but is idealised as a parameter (*n_pred*); *i.e.* it is constant for the duration of a simulation run. Similarly the effect of the growth process which in reality causes the weight of the grass to increase is idealised as an intermediate variable rather than a state variable. Thus, the growth process is not directly modelled. The value of the attribute *grs_wt* which depends

on dry season rainfall also happens to remain constant. That is not because it is a parameter; rather it is because ultimately the only variables that are used to compute it do not depend on exogenous or state variables, but only on parameters.

Explicit interest in the wildebeest migration may be noted using the appropriate *_interest* construct, as well as the fact that for the purpose of this simulation, it will be ignored. This is achieved by the following specifications:

$$role\_def(migration, migrant, animal^{\odot})$$
$$role\_esys(wb\_migration, migrant, wb\_pop)$$
$$occ\_interest(wb\_migration, manual, ignore)$$

manual means that this specification was created manually, not automatically. Automatic creation of *_interest* specifications is not yet implemented. In the future, we intend to infer *_interest* specifications from goals, because users are necessarily interested in things mentioned in goals. We distinguish between manually and automatically noted interest because the latter may disappear quietly if, say the goal is retracted. Users will be asked to confirm removal of manually created ones.

Implicit Idealisation

Implicit idealisations correspond to anything that is in the description of the ecological system, but is not in the runnable model. This is useful to someone who is interested in the details of the ecological system being modelled, but may not have constructed the model or constructed it a long time ago. As an example consider the aggregated predator population consisting of the lion and hyena populations. No interest is noted for either of these populations; they do not occur in any goals; nor do they give rise to any model variables. Thus, they are not directly represented in the simulation model. However, because there is an explicit record of the aggregation, (using $\subset$) we can recover an idealisation decision. Most of the extra ecological information that we discussed in § 6.7.3 serves to record implicit idealisation decisions of this kind. Additionally, it is useful because it makes it easier to expand and/or experiment with the runnable model.

The very fact that something is explicitly encoded in the ecological system description means that the person who put it there thought it was *potentially* important (*i.e.* someone would be interested in it) for *some* future simulation modelling exercise by themself or someone else. The same ecological system de-

scription may be used to construct many different simulation models concentrating on different aspects. Note that the *_interest* specifications do not denote this general sort of interest, but more specifically, interest with respect to one simulation exercise. Thus, if different simulation models of the same ecological system are to be saved and retrieved, the *_interest* constructs should be saved along with the runnable-model constructs to help document it. This merely reflects that fact that the simulation modelling information level contains the dialogue level and the runnable-model level; thus the goals and user-specified defaults also get saved as part of the simulation model.

6.9 Conceptual Distance

A major design constraint for the formalism is to keep conceptual distance to a minimum for end users of ELK. In practical terms, this means that the user interface commands must be readily understood in ecological and modelling terms. At no time can we expect users to examine or manipulate logical expressions directly. There are two related issues here:

1. designing user interface commands
2. bridging the conceptual gap

The easiest way to facilitate development of easily used interface commands is to design constructs whose semantics are expressed in terms that users are familiar with. This we have done, but it is not enough. The conceptual gap we are dealing with is considerable. A set of differential equations which we ultimately construct is a rather different sort of thing than lions eating wildebeest. We have undertaken, therefore, to design a series of constructs which form a conceptual bridge from the terms and concepts about ecology, to the terms and concepts in simulation modelling. The bridge is manifest in our four level knowledge ontology.

To illustrate how this bridging works, consider the sequence of constructs in figure 6–6, presented in an order which could very well correspond to the events in an actual elicitation. We concentrate on wildebeest, although many other instantiated constructs are required in conjunction with those listed. A user begins by modifying the sort hierarchy if necessary. They proceed by expressing interest in certain sorts, and create corresponding entities. They may express goals involving

aspects of these entities; this forces them to identify what attributes of the entities are important and thus likely to give rise to model variables. They may have specific ideas about how certain classes of variables should be idealised when they are created, and/or cause them to be created automatically. They must decide what processes are affecting the attributes of the important entities. These are likely to give rise to effect variables. They must create the model variables corresponding both to attributes and effects. They must decide how to idealise them, if different from the defaults specified either by themselves or the system. Eventually, a set of differential equations is implicitly specified.

So, we begin with general ecological concepts, and gradually by taking relatively small steps, we elicit the structure of the model in terms of differential equations. The conceptual bridge formed by this series of constructs is what allows us to use the technique of *gradual elaboration* in eliciting models. We come back to this point in chapter 7.

There are three distinct modelling (*i.e.* idealisation) phases required for using ELK. Each embodies a considerable degree of idealisation and results in a description in one of our information levels. The phases and intermediate descriptions are summarised as follows:

- real world (not represented!)

  Phase I: [real world] idealised as:
- general/ecological knowledge base

  Phase II: [conceptual model of the world] idealised as:
- ecological system description

  Phase III: [description of the ecological system] idealised as:
- runnable model

Note that there is not a strict temporal relationship implied by the ordering of these phases; rather there is considerable interleaving. This is evident in figure 6–6.

Phase I idealises the real world. We include whatever we can that is likely to be required in some ecological model and ignore everything else. For example, we must include animals, but Freud's theories of psychoanalysis are of no relevance . This phase further divides into two stages. The first is a prior system development phase which does not involve end users. This resulted in our Process/Entity/Attribute/Value *conceptual modelling framework* described in general

1. A wildebeest is a sort of animal.
   $wb \sqsubset^s animal$
2. Interest in wildebeest is noted.
   $sort\_interest(wb)$
3. There is a wildebeest population in the ecological system of interest.
   $wb\_pop{:}\#wb$
4. Interest in the affect of an increase in the dry season rainfall on the size (in numbers) of the wildebeest population is noted.
   $xy\_affects(dry\_ssn\_rain, increase, \lambda T{:}time.number(wb\_pop, T))$
5. By default, the *number* attribute of all collections of animals in the ecological system (if they are to be included in the model) shall be idealised as a state variable on positive numbers rather than naturals.
   $att\_inh'(number, \#animal, positive, stvar, manual)$
6. The number attribute of the wildebeest population gives rise to a state variable idealised on positive reals. *n_wb* is the name of the variable.
   $att\_var'(number, wb\_pop, n\_wb, positive, stvar)$
7. The population of predators is preying on the wildebeest population thereby tending to decrease their numbers. Predation is not observable on a time scale of less than one day.
   $role\_esys(wb\_pred, prey, wb\_pop)$ $role\_esys(wb\_pred, predator, predators)$
   $effect\_esys(wb\_pred, number, prey, decrease, day)$
8. The decrease in numbers of the wildebeest due to the occurrence of predation *wb_pred* is idealised as a partial rate variable which contributes a negative amount to the net annual rate of change of the number of wildebeest. *wb_eaten* is the name of the variable.
   $effect\_var(wb\_pred, number, prey, wb\_eaten, decrease, year)$
9. ...which implicitly specifies:
   $prate\_variable(wb\_eaten, positive, na, n\_wb)$
10. ...which in conjunction with various other specifications just like these implicitly defines the following differential equation:
    $\forall T{:}time.rate(n\_wb)(T) = -wb\_die(T) - wb\_eaten(T) + cf\_surv(T)$

The above sequence of instantiated constructs shows how we bridge the gap between very general ecological concepts and very specific simulation modelling concepts. The key to this bridging is the three distinct steps in the idealisation process corresponding to the specification of the general/ecological knowledge base, the specific ecological system to be modelled, and the simulation model. This sequence shows that although conceptually distinct, these phases may be interleaved during model elicitation.

Figure 6–6: Bridging the Gap

terms in chapter 2; the theory in chapter 5, and its implementation in this chapter. The result of this first stage is a set of (uninstantiated) constructs for representing processes, entities, attributes, and values. The second stage in the first idealisation phase entails creating a substantial number of instantiations of these constructs. This includes (a) building sort and part hierarchies, (b) assigning attributes (with value spaces) to these sorts (and to induced collection types) (c) creating and characterising what processes there are and their associated effects. Part of this is the task of system developers in conjunction with expert ecologists; part is the task of each ELK user not all of whose problem specific requirements will already be catered for. The final result of the first phase is the general/ecological knowledge base which may be thought of as a conceptual model of the [ecological] world.

Phase II further idealises the conceptual model of the world embodied in the general/ecological knowledge base. Any particular ecological system that one chooses to describe will use only a portion of the general/ecological knowledge base. Relatively few of the entity types in the hierarchy will have explicit entities created. Of those that do, most of the substructure will not be explicit. For example, set entities are created with no individuals or subsets represented; individual entities are created with no explicit parts represented. Only some of the processes that are occurring will be represented. This second phase results in what we refer to as the description or conceptual model of the ecological system.

Phase III entails further idealisation of this conceptual model and results in the runnable simulation model. Here we ignore many attributes, and simplify others by modelling them with different value spaces than they really have. In ignoring attributes, we also ignore many actual effects of processes. Additionally, we may model these effects in ways that are simplifications of what is really going on. What substructure is defined explicitly in the ecological system description, may be ignored in the simulation model (*e.g.* the lion and hyena populations).

The dialogue level provides an intermediate link between the ecological description and the runnable model, but is distinct from both. As a link into the simulation modelling phase, users may express goals or interest in certain things. They may define their own defaults which partially automate the idealisation decisions that they need to make.

6.10 Conclusion

In the beginning of this chapter we noted four key issues that we faced in designing ELK.

- model comprehension
- expressive power
- conceptual distance
- assistance during elicitation

The second was addressed in chapter 5, but only from a theoretical point of view. Also, discussion of certain important predications was excluded. In this chapter, we have completed our treatment of this issue by introducing the dialogue level constructs and discussing important implementation issues for the constructs at all four levels in our knowledge ontology. The important things to note about the relationship between the theory and the implementation are:

- The object/meta level distinction is different from the distinctions in our four level knowledge ontology. The object-level is used both to describe ecological systems *and* models. Also, the $\ldots var\_fn$ functions explicitly bridge these two levels.
- Users must dynamically alter the object-level language in the process of building the general/ecological knowledge base, creating entities and creating model variables.
- To do this, there are a series of meta-level constructs which are used to create object-level sorts, entities functions, and relations.
- The meta-level constructs are used to distinguish between object-level functions whose types are identical, but which are in different levels in our ontology.
- The technique of *implicit specification* is a key aspect of the theory as well as the implementation.

We have described the meta-level constructs used to define the dynamic parts of the object-level language. We have also given the details of how certain key inferences are accomplished. The use of implicit specification is evident in many ways as we noted. The main techniques we use to achieve expressive power are:

- rich typing
- few reusable primitives
- combining functions

As a vehicle for testing our knowledge ontology hypotheses, we have classified the constructs into one of the four categories in our knowledge ontology. These are:

- ecological information
  General/ecological knowledge
  Description of ecological system
- simulation modelling information
  Dialogue
  Runnable model

The *_def* constructs are used to create object-level functions and relations that comprise the general/ecological knowledge base (in addition to the system created ones including $\sqsubset^s$, $\subset^p$ etc.). The *_esys* constructs are used to specify information at the ecological system level (in addition to the system-created relations including :, $\subset$, etc.). The *_var* constructs are used directly to create object-level functions that correspond to the variables in the runnable model (there are no system-created functions at the runnable-model level). The *_variable* constructs are defined implicitly in terms of the *_var* constructs. They explicitly blur the distinction between pure and ecological model variables which is unnecessary for the purpose of representing the runnable model. We showed how the differential equations are automatically generated from the *_variable* specifications. We also described the representation for non-differential equations used for computing intermediate, partial rate, and exogenous variables. Finally, we described how the runnable-model specification may be compiled and the simulation run.

The *_inh* and *_interest* constructs as well as those for representing goals are used to specify information at the dialogue level. These do not correspond directly to any object-level entities, functions or relations for either ecological level or the runnable-model level. Instead they are meta-level constructs expressed in terms of these object-level entities, functions, and relations as well as various instructions to the dialogue manager.

We have demonstrated a significant amount of the expressive capability of our formalism by illustrating the use of these constructs in the context of the

Serengeti ecosystem. In doing so, we have begun to seriously test our *ontology completeness hypotheses* which says that all information relevant to the ecological modelling process can unambiguously be placed into one of our four categories. In formalising the Serengeti example, we have achieved a very high percentage. One interesting example showing the power of the representation is the rather complex concept of specific rate of predation. The constructs we have are certainly not perfect however. For example, our representation for processes with two or more agents is fairly weak. Consider predation: we are not able to specify which types of animals may prey on which other types; nor can we distinguish between predation among individuals and predation among groups. Also, it might be better to represent attributes like *spr_pred* and *fecundity* explicitly in terms of the associated processes (*e.g.* predation and reproduction, respectively).

We have also begun to seriously test our *ontology usefulness hypothesis* which says that our four level ontology is useful. The main uses correspond to the latter three major issues listed at the beginning of this chapter repeated at the beginning of this section. These are:

- to reduce conceptual distance by bridging the gap
- to facilitate model comprehension
- to assist in the elicitation of formal descriptions of ecological knowledge, systems, and models.

We have considered the first two; the third is the subject of the next chapter.

Chapter 7

ELK: Elicitation

7.1 Introduction

This is the third and final chapter concerned with describing our solution (embodied in ELK) to the reformulation of the ecological modelling formalisation problem. In chapter 4 we noted many specific issues and requirements that arise in the context of the specific formalisation problem of ecological modelling. The chief ones are:

- model comprehension
- expressive power
- conceptual distance
- assistance
    - identification and control of the various idealisation search spaces
    - consistency checking
    - no redundancy
    - flexibility
    - relief from redundant and/or menial tasks

The basis for addressing the first four is distinguishing between ecological and simulation modelling information and bridging the two. The chief support for model comprehension comes from the bridging between the two levels. The same bridging is also the key to reducing conceptual distance. We also presented a language for expressing the required information. What we have not done is demonstrated that an interface can be constructed that allows complex logical terms to be constructed easily and safely.

It is unwise to expect ecologists to be able directly to create, read and understand expressions containing complex logical terms using the notation of lambda calculus, higher-order functions, etc.. The meta-level constructs that are manifest in the implementation as Prolog terms and predicates do not constitute a viable alternative. The major unfinished business in this thesis is to demonstrate that it is possible to embody the material presented in chapters 5 and 6 in a 'user-friendly' computer assistant.

We require a good dose of syntactic sugar to achieve this. The main purpose of this chapter is to explain the interface design and illustrate how ELK is used to construct models.

We first give an overview of the basic facilities provided by ELK and discuss the basic principles for using it. We then describe in turn the three phases in idealisation discussed in § 6.9. We illustrate how to build the general/ecological knowledge base, the description of the ecological system, and the description of the simulation model. Where appropriate, we show how the dialogue level constructs may be used to assist in the elicitation process.

We conclude by discussing each of the above requirements and review the techniques used to meet them. This completes the substantiation of the claims made with respect to the design rationale in chapter 4.

7.2 Overview of Elk

Usually, a complex formalism is inherently difficult to use, or at best requires considerable training for one to become adept at encoding things in it. A good example of this is first order predicate logic, which is notoriously difficult to use directly. Competence requires considerable training and practice in logic and knowledge representation. The rich type structure and higher-order functions make Elklogic much more complex than first order logic. However because we are only representing information in our constrained PESAV framework[1], the task is manageable.

The constructs have been carefully designed with the interface requirement in mind. In most cases, the interface is fairly straightforward. There have been some significant challenges, however.

The major factor which simplifies the interface design is the fact that the semantics of most constructs are expressed in straightforward ecological and/or modelling terms. Where complex compound structures are built up, there are specific rules constraining how the primitives may be combined, and rules for interpreting the ecological meaning of the compound constructs in terms of the meaning of the primitive ones. These rules derive directly from the typing.

7.2.1 Interface and Facilities

Commands

Figure 7–2 shows the main ELK interface. There are two major classes of commands EDIT and DISPLAY. Edit commands are used to add, remove, or modify some part of the specification (at any of the four levels). Display commands are used to examine some part of the specification. There are currently well over a hundred commands available. Some of the edit commands are:

- A new sort may be added to the sort hierarchy.
- Instances of entities may be created.
- Attributes and model variables may be created.

Some of the display commands are:

---

[1] Recall this stands for Process/Entity/Substructure/Attribute/Value.

- Any of various hierarchies may be displayed (*e.g.* sort, part, component, possible component)
- A user may find out all the attributes that apply to a given entity type.
- A user may get a listing of all the currently defined state variables, and their ecological meaning,

For each major class (*i.e.* EDIT and DISPLAY) there is a menu bar[2] containing several command categories. For example, the TAXONOMY display category is used to display any of the hierarchies. Clicking on one of the category labels causes a tree-structured walking menu to appear. The leaves for each category correspond to individual commands. For example, the ATTS category (in the EDIT class) has two choices in the top level menu corresponding to the general/ecological and modelling levels. The former splits into three further options for adding, modifying, or removing attributes in the general/ecological knowledge base. The latter splits into two categories, one for specifying defaults, the other for defining variables and parameters. Each of these further splits into two options: add and remove. This is depicted in figure 7–1. When referring to menu options in the text, we use small capitals (*e.g.* GEN/ECL). To show the path taken through a walking menu to select a command, we use the following notation illustrated by example: ATTS–GEN/ECL–ADD.

Note that this menu structure for attributes directly reflects our ontology. The MODEL option corresponds to the simulation modelling layer which further divides into the specify defaults part of the dialogue level and the variable definition part of the runnable-model level. There are separate command categories for specifying other specifications at the dialogue and runnable-model levels. For example, all the interest specification constructs come under one heading (INTEREST). There is also a separate MODEL edit category for creating pure model variables, selecting pure schemata for computing variables, specifying the model output, compiling and running the model.

Note also that in most cases these interface commands correspond exactly to the constructs presented in chapters 5 and 6. For example, the ATTS–MODEL–SPECIFY DEFAULTS-ADD command creates an instance of the *att_inh* construct;

---

[2] This is a term used in the Quintus Prowindows Manual.

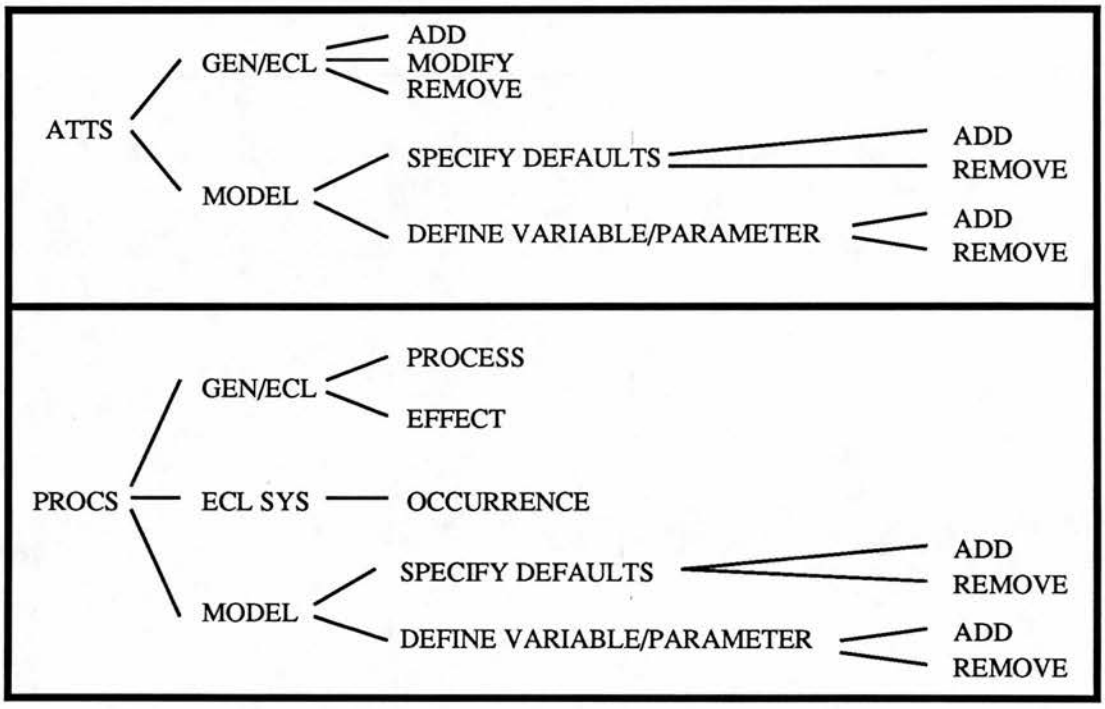


Figure 7–1: Commands for Editing Attributes and Processes

the PROCS–MODEL–DEFINE VARIABLE/PARAMETER–REMOVE command removes an instance of the *effect_var* construct. This is a crucial point. The constructs have been carefully designed this way to make the interface relatively easy to implement and simultaneously ensure that the end user understands the interface in ecological terms.

Selecting a command does not execute it. Rather, the system displays information about the command. This includes a one line command description, instructions or comments explaining how to use the command and displaying what is essentially a frame with various slots which are used as inputs to the command. For example, if a user decides to add an attribute, slots appear for the name of the attribute, the sort to which it applies, its value space, etc. This is the syntactic sugar for instantiating the *att_def* construct. There is an interface slot for each argument. The instructions explain what to do for each slot, and give warnings if the selected command is highly destructive (*e.g.* clear the whole knowledge base!). When a user is satisfied that the inputs are correct they click the OK button which executes the command. If at any time before they do this, they decide they wish to do something else instead, they may click CANCEL, and select another command.

There is a separate interface for goal acquisition. Currently, it is only partly linked with the main command interface. Historically, the goal acquisition subsystem was implemented first. Although based on the same theory, the underlying implementation needs to be properly integrated with the main program.

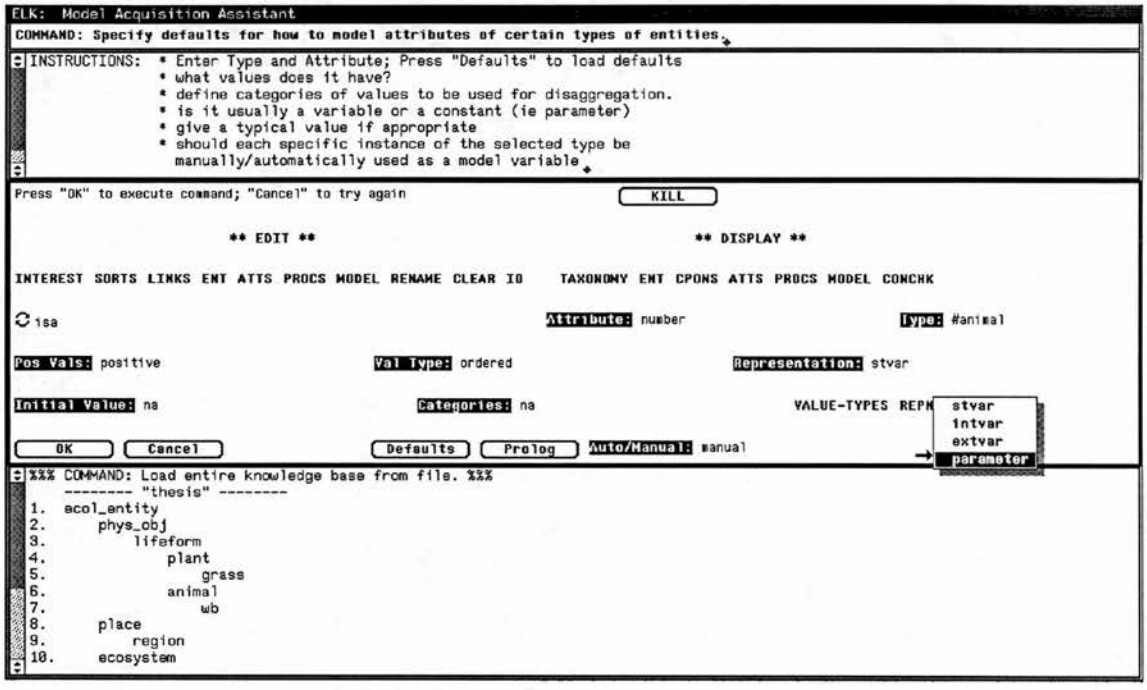
In addition to the edit and display commands, there are also IO commands for saving and loading models. We elaborate on this in § 7.2.2.

Messages

To keep the user informed of what is going on, and to provide a record of a session, the system produces a copious quantity of messages, each specific to a particular situation.[3] There are 3 main types:

---

[3] There are a few hundred situations currently catered for. The full text has been provided for 25-50% of these; the rest result in a coded message which is adequate for debugging purposes, but not for users.



The user specifies that by default every variable corresponding to the attribute 'number' of type '#animal' is to be idealised as a parameter with a value space of positive reals. They first click on the ATTS *edit command category; then they 'walk' through the menu that pops up to select the current command. Instructions for its use are shown, as well as the slots that must be filled in before command execution. A portion of the sort hierarchy is shown below from a past use of the display taxonomy command.*

Figure 7–2: ELK Command Interface

OK: These are routine confirmation messages that are printed after execution of each command (or command step within a multi-step command). For example, when the attribute *weight* is created, the following is printed:

```
COMMAND: Define attribute for a sort.
***** OK *****
"weight" is a newly defined attribute for a phys_obj.
--------------
```

Warning: These are given in situations where there is no specific problem, but the system deems it worthwhile to bring something to the attention of the user. For example, if the part link between *paw* and *lion* is removed, the following message might get printed (after the OK message confirming that the link was removed):[4]

```
*** Warning ***
The sort "lion" is not being used.

*** Warning ***
The sort "paw" is not attached to the "isa" taxonomy.
```

Error: These are given when a user has given illegal input for some command which therefore cannot be executed. These are almost always due to typing problems of some kind. Wherever possible, useful comments are provided to help the user identify the problem. For example if inadvertently a user tried to say the predator population was a component of the lion population, the following error message is printed:

```
*******  Error  *******
Currently, a animal-collection is not known to be a valid
component  of a lion-collection.
If you think it should be, then update the part and/or isa
taxonomy accordingly.  To assist in this regard note the
following:

The objects that a animal-collection can be a direct
component of are:
    animal-collection

The objects that can be a direct component of a lion-collection
are:
    lion
```

[4] In the implementation, the sort hierarchy is called the 'isa taxonomy'.

```
lion-collection
************************
```

7.2.2 Using ELK

The development of the PESAV conceptual modelling framework effectively defines the search space for the specification of the general/ecological knowledge base. Creating such a knowledge base is very much a domain dependent exercise which we have not carried out to any great extent. This is partly because it is an extensive task in its own right, but more importantly, we expect that each ecologist will prefer to tailor their general/ecological knowledge base to their own needs. They may wish to use the names of their choosing, rather than those of other ecologists. They will not want the sort hierarchy to be cluttered with many sorts that are unlikely to ever be used by them; they will not want to be subjected to menus with lots of irrelevant attributes. For this reason, we provide only the most basic sorts and attributes. Except for an immutable kernel of system sorts, a user may if they see fit start virtually from scratch. We anticipate that the construction of the bulk of the general/ecological knowledge base will be a task done separately from describing various ecological systems and simulation models.

There is a notional continuum of what we call *problem specificity* of information. For the purposes of this discussion we take this to be equivalent to the opposite of 'likely frequency of use' of information. Thus, information that is highly problem specific is unlikely to be used except for that specific problem. Conversely, information which is not at all problem specific, is likely to be used for a wide variety of problems. As far as the general/ecological level is concerned, at the one end of the spectrum, we have extremely general knowledge quite independent from ecology (*e.g.* parts, sets, numbers, attributes). This gradually changes to knowledge which is specific to the ecology domain, but is quite general for that domain (*e.g.* animals, water, biomass, sex). Moving along the continuum, we get to specific sub domains within ecology, such as population dynamics, or forestry. Finally, there are highly problem specific concepts such as some rare animal species. Some problem specific concepts need not be represented in the general/ecological knowledge base at all. Whether they do depends on whether the concept will be used frequently for that problem. The design for ELK embodies three distinct points along this continuum:

1. *general:* an immutable kernel
2. *domain specific:* a permanent portion,
3. *problem specific:* a user modifiable portion

The idea is that different portions of the specification can be saved and retrieved independently. We describe how this is meant to work below. Currently, it is only partially implemented.

Although designed to represent ecological and modelling information, the core language concepts are not specific to ecology, or even modelling. This domain independent kernel includes two main aspects. First, there are the language constructs as described in chapters 5 and 6. These include the relations for representing the sort and part hierarchies, attributes, the set formation operator, and some very basic sorts and attributes (*e.g. entity, number*). Secondly, along with this is a large amount of general machinery which manipulates and guides the operations that can be performed on these constructs. This includes the rules for inheritance, etc. For example, the typing of the higher-order functions like *average*, *rate*, *qnam*, defines specific rules for combining and nesting them. For instance, if *biomass* is an attribute of *sheep* then $qnam\_ent(average, biomass)$ is an attribute of $set(sheep)$.

The immutable kernel may only be changed by the system developers. This would generally only be necessary when new expressive power is added.

The permanent portion corresponds to the bulk of the general/ecological knowledge base which will be of general use for some class of users. It will enable a reasonably wide range of ecological systems to be specified in a certain domain. As noted above, this part is created separately from the normal modelling process. The idea is that a special mode will be required to edit the permanent knowledge base. In normal use, the contents will not be removable. Note that removing things from the permanent knowledge base may cause the previous ecological systems built based on it to be inconsistent (as well as any simulation models based on any of those ecological systems). Introducing new versions of any software causes these kinds of problems.

The user modifiable portion will typically be specific to individual ecological systems, but not used frequently enough to be put into the permanent knowledge base. We must allow for migration of concepts from the user modifiable to permanent portions if it later turns out that concepts are frequently used. This

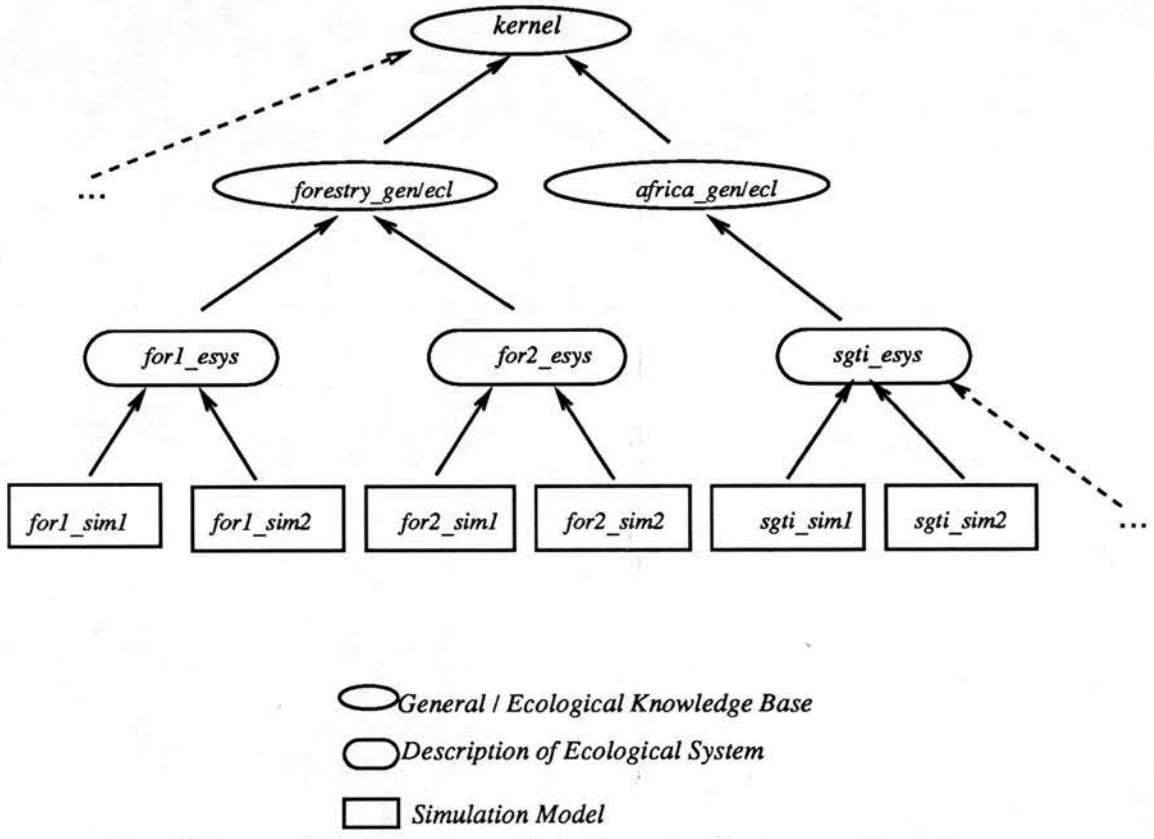
introduces more version control problems. Every existing ecological system described using the previous version of the permanent general/ecological knowledge base may need updating.

There may be any number of different permanent knowledge bases for different sub-domains. It will be possible for these to be saved and retrieved independently, each embodying the immutable kernel. Similarly, there may be any number of different ecological systems corresponding to a single permanent knowledge base. An ecological system created using one permanent knowledge base may not be loaded with a different permanent knowledge base. Finally, any number of simulation models may be created as different idealisations of a single ecological system. Similarly, a simulation model which idealises one ecological system may not be loaded with a different ecological system. This gives rise to the tree structure of models that may be constructed with ELK illustrated in figure 7–3. This is one important way that we can achieve the requirement of *reuse*.

Note that, *the distinctions on the problem specificity continuum* (general, domain-specific, problem-specific) *are not the same as the distinctions in the knowledge ontology* (general/ecological knowledge base, ecological system description, and runnable model). For example, the general/ecological knowledge base contains general concepts (*e.g. number*), domain specific ones (*e.g. tree*), as well as problem-specific ones (*e.g. 7_toed_tree_lizard*). Furthermore, concepts can migrate from the problem-specific level to the domain-specific level when/if a user or group of users decides that it is likely to apply to more than one problem in a domain. This can happen even though the concept is represented exactly the same way. Such migration of concepts does not make sense between the different levels in the knowledge ontology.

In the succeeding sections, we give the important details with respect to the interface for each major command category. We proceed by considering in turn each of the three idealisation phases introduced in § 6.9. They correspond to the different levels of information in our ontology. For each phase, we give a brief overview of how each major command is used. We mention and/or illustrate assistance relevant only to specific commands, when we introduce the command. We discuss more general kinds of assistance that apply to a range of commands, in a summary at the end of this chapter.

After all the commands have been discussed, we turn the discussion around



It will be possible to use ELK *to independently save and retrieve:*

- *different versions of the general/ecological knowledge base based on the kernel*
- *descriptions of different ecological systems based on the same general/ecological knowledge base*
- *different simulation models based on the same ecological system*

Figure 7–3: Hierarchy of Models

and focus on the key benefits and techniques with respect to interface assistance. We mention various more general examples of benefits that apply to more than one command. This substantiates the claims we made with respect to the design rationale described in chapter 4. Keep in mind that although conceptually distinct, these phases are not to be viewed as strictly temporally ordered. Quite the contrary, ELK had been designed to allow maximum flexibility for user to do things in any order they wish. Thus, there may be considerable interleaving.

7.3 General/Ecological Knowledge Base

We describe how a user may construct the general/ecological knowledge base which constitutes an idealisation of some particular ecological domain cast in the PESAV framework. There are four main jobs:

1. define a sort hierarchy
2. define a part hierarchy
3. create and characterise attributes
4. create and characterise processes

7.3.1 Sort and Posible Part Hierarchies

There is a uniform mechanism for editing and displaying hierarchies. A user clicks on TAXONOMY, and chooses from a menu of hierarchies. These include the sort, possible part, possible component, and component hierarchies. The latter is part of the ecological system level considered later. The possible component hierarchy may be displayed, but because it is wholly induced, it may not be edited directly. If the user wishes to edit the part hierarchy, the slots appear labeled PARTS and COMPOSITE. If the sort hierarchy is selected, the labels are SORT and SUBSORTS, etc. More than one link may be added at the same time., but only to a single parent node. A node may also be linked in between two existing nodes.

Consistency Checking

The system gives a warning if a link is already there, or if a child node is already linked to a different parent (*i.e.* the relation is a graph, not a tree). If a link is added to the part hierarchy, and the sorts are not yet in the sort hierarchy, the

user is informed that new sorts have been created. When links are removed, a warning is given if the sort is no longer in use (*cf paw/lion* example above.).

Relief from Redundant/Menial Tasks

The system automatically keeps track of what the sorts are from the contents of the part and sort hierarchies. The system induces the transitive versions of each of these hierarchies. It induces the component hierarchy entirely. Thus, with a minimum of effort, a considerable amount of useful specification may be achieved.

Transparency

The ability to browse through the hierarchies enables users to see quickly the results of what they have specified. This is especially useful with respect to the possible component hierarchy which is not directly specified (see figure 5–3, page 156). Users may check that certain links are there that should be and vice versa. They may also obtain text explanations of the nature of a particular $\subset^p$ link. For example, if *claw* $\prec^p$ *paw* $\prec^p$ *cat*, and *lion* $\sqsubset^s$ *cat*, the system explains why *claw* $\subset^p$ *lion* as follows:

```
The reason a claw can be a component of a lion is:

A claw can be a part of a paw.
  **AND**
A paw can be a part of a lion
    because a lion is a cat.
    and a paw can be a part of a cat,
```

7.3.2 Attributes

In § 7.2.1 we outlined the interface for creating and characterising attributes. One thing we did not mention there but discussed in chapter 6 was the distinction between basic and modified attributes. Suppose *age* was created as an attribute of physical objects. At that level, there is no generally useful age categories that might be used to define substructure. However, for the subsort *lion*, there is, namely $\{cub, adult\}$. This may be specified as follows. First, the user selects the edit command ATTS–DEFINE–MODIFY. They then enter age and lion in the appropriate slots. At this point, they may request a brief explanation about this

attribute by clicking the EXPLAIN button. In this case, it would produce the following output:

```
Attribute "age" for a lion is inherited from a phys_obj.
Attribute "age" is a basic attribute for a phys_obj.
It is user defined and may be changed at will.
```

This would be slightly different if age had already been modified for an intermediate sort. If the user mistypes, or forgets to enter the attribute, the system issues an appropriate message. At this point, assuming the user wishes to go through with the modification, they click LOAD ATT causing the current specifications from *phys_obj* to be loaded. These are then modified as appropriate. The interface for this is a very similar to that depicted in figure 7–2.

Removing attribute definitions is easier; it only requires entering an attribute and a sort.

Consistency Checking

Some of the attribute related checks include:

- is attribute being assigned to a sort which does not yet exist?
- is an attribute already defined?
  - should not be if adding new one
  - should be if trying to modify or remove it.

Other consistency checking that is not implemented yet entails ensuring that *cub*, etc are ranges in the appropriate value spaces. For example, it might be that a cub was defined to be up to one year old. That is:

$$cub = \{[0, 1]\}{:}set(positive)$$

The typing makes this easy to do.

Relief from Redundant/Menial Tasks

The inheritance of attributes from sorts to subsorts as well as the induced attributes based on *average*, etc. greatly reduces the amount of explicit specification required. Also, when attribute descriptions are being modified, the system loads the defaults, saving the user the bother.

Transparency

There are various facilities for examining the current state of the attribute definitions. We have already seen that brief explanations may be provided indicating whether an attribute is basic, modified, or inherited. Concise summaries may also be obtained of all the basic and/or inherited attributes of some/all sorts. For instance, assuming the knowledge base is defined for the Serengeti, selecting the display command: ATTS–ALL–OF CERTAIN SORTS and entering "animal" in the appropriate slot, gives the following:

```
COMMAND: List all basic and inherited attributes for selected sorts

LISTING ALL BASIC AND INHERITED ATTRIBUTES FOR A ANIMAL.
** The basic attributes of an animal are:
      cap_cf, r_surv, htime, fecundity & cap_cf
** A animal inherits attributes from the following:
      lifeform, phys_obj, ecol_entity & entity
   From a lifeform, it inherits:
         biomass & amount_ddt
   From a phys_obj, it inherits:
         weight & age
```

Note that we do not list the induced attributes. This is mostly to reduce clutter. Note also, the attribute *amount_ddt*. There could be dozens or hundreds of such attributes. Which ones to include would be up to each user. It would be possible to extend the logic to incorporate the notion of *amount*. For example, *amount*(*ddt*), *amount*(*water*) etc could be induced attributes whose existence derived from the sorts *ddt* and *water*. This would require special machinery and could only be achieved by system programmers.

7.3.3 Processes

There are two stages in specifying process information at the general/ecological level (see figure 7–4). The first is to create the process. This is done with the PROCS–GEN/ECL–PROCESS-ADD command. The user first enters the name of the process (*e.g. grazing*). Then, one or more roles are characterised. Each role is given a name (*e.g. grazer*), and a type restriction (*e.g. animal*). Finally, the user indicates what attributes are affected by this process. For distinguishing whether the attributes are necessarily or possibly affected (*e.g. biomass* versus *amount_ddt*), there are two separate attribute lists. This first stage results in

up to three *role_def* specifications, and an intermediate construct to record the affected attributes (*affected_atts*).

The second stage consists of specifying one at a time the details of each effect; there is one for each affected attribute noted in the first stage. Indicating the process, the role, and the affected attribute identifies which effect is being characterised (*e.g. grazing, grazer, & biomass*). The user must then specify the nature of the effect, and the time scale over which the process is relevant (*e.g.* increase, hour). After specifying this and pressing the OK button, and the command is executed. If there are problems the system prints warning and/or error messages. If the are no errors, the following instantiated construct is added to the specification.

$$effect\_def(grazing, biomass, grazer, increase, hour, necessary)$$

Note how the slots in the interface correspond directly to the arguments in the corresponding constructs. Note also, that the user need not specify here whether the attribute is necessarily/possibly affected. This has already been indicated in the previous stage. This slot is included in this stage only for convenience; when the user selects the process, and attribute, ELK fills in the value of this slot automatically; the user may not modify it here.

Search and Consistency Checking

ELK provides extensive assistance for creating processes. Except occasionally where it is not possible, menus are presented to the user for every slot. In some cases, heuristics are used to prioritise the choices so that the most likely ones come first. The default is usually alphabetical order. This reduces the amount of time required to execute these commands. Little typing[5] is required, but more importantly, a user's memory is not taxed, freeing up their minds to concentrate on the important things. Also, extensive consistency checking facilities are in place.

The menus are dynamically generated based on the existing state of the specification and general type consistency rules. For creating a process, the user must enter the name of the process and of the roles. For the role type slot, ELK generates a menu including all the sorts. For the attribute slots, ELK generates a list of all the attributes that apply to the types of the entities that may participate in either

[5] That is, using the keyboard.

role. Thus, if *animal* and *plant* are entered in the type slots for the grazer and grazed roles, the list will include *biomass*, *weight*, etc but not *area*. The heuristics used to order attributes are:

- First priority to attributes that the user is interested in.
- Second priority to attributes that apply to the types closest in the type hierarchy to the role types in the type restriction slot(s) for the role(s).

Thus, unless interest had been specified in the latter, *biomass* would come before *weight*. This is because the most general type to which *biomass* applies (*lifeform*) is closer to *animal* and *plant* in the type hierarchy than the most general type to which weight applies (*phys_obj*).

Currently, 'interest' only exists by virtue of the user explicitly noting interest in the attribute for some type. However, there is much scope for inferring interest indirectly. For example, if interest is explicitly noted in biomass of *ln:lion* then interest is indirectly noted in biomass of lions. Interest is also indirectly noted if an attribute is mentioned is some goal (*e.g.* "What is the affect of temperature on biomass?".

The generation and prioritisation of attributes is a general facility useful not only for processes, but in other situations as well. We shall mention some of these in subsequent sections.

When specifying effects, the user need not use the keyboard at all, there are menus for every slot. Before a process has been selected, the menus for role and nature of effect are not activated; instead, the user is reminded to choose a process first. Once chosen, the role menu is activated, but the attribute menu is still not; it now reminds the user to select a role first.

The interface for processes is the most thoroughly implemented aspect of ELK. It illustrates the intimate connection between rich type structure, pruning the specification search space, and maintaining consistency. By using type information to generate menus all of whose choices are guaranteed to be consistent we simultaneously prune the specification search space and ensure consistency. Note that the 'type checking' is done *before* the user gives the specification. There is scope for doing this for virtually every command in ELK. However, this is a highly time consuming process and only with respect to processes has ELK used this to full advantage. Compared to the commands where the menus are not generated, there is a dramatic increase in speed of operation.

Process:	predation	
Role:	predator	animal
Role:	prey	animal
Necessarily Affected Attributes:		number,biomass
Possibly Affected Attributes:		amount_ddt

Process:	predation
Attribute:	biomass
Role:	predator
Nature of Effect:	transfer
Time Scale:	second
Necessary/Possible:	necessary

The user defines the process of predation is two stages, using a different command for each. First, the name of the process, the roles and affected attributes are given. Then, one by one, they define the effects. Menus for each choice are provided. The necessary/possible slot in the second stage is shown for completeness; it may not be altered except by modifying the first stage.

Figure 7–4: Creating the Predation Process

7.4 Ecological System Description

We illustrate how users may describe the ecological system of interest. This constitutes an idealisation of the general/ecological knowledge base which is itself an idealisation of some particular ecological domain. There are three main jobs:

1. create entities (individuals and collections)
2. specify substructure between these entities
3. specify occurrences of processes

7.4.1 Entities

Entities are created by selecting the edit command ENT–ADD. Several entities may be created at once, by giving a list of entities of one type, and/or a list of $E{:}T$ pairs (see figure 7–5). As discussed in § 5.6.5 we do not allow entities to be created using *set*, only #. This is because (a) the details of the set substructure dynamically change, and (b) it is usually unnecessary to know what the details

are. Thus, the user need only distinguish between individuals and collections. This is one way that ELK hides the underlying complexities.

We provide a simple notation for creating multiple numbers of an entity using the indexing technique in a purely syntactic fashion. If we wish to create 30 wildebeest ($wbst(1)$, $wbst(2)$, ...), then a user simply enters "*wildebeest##30*" in the name slot, and *wb* in the type slot. Logically, this results in the following being added to the specification: $\forall I \in I_{30}.wbst(I):wb$. N.B. ## notation is not part of the specification language, or the logic, just the interface.

A special notation is used for specifying time substructure. For example, specifying "$yr(day, hour)$" in the name slot and *year* in the type slot results in the creation of a year with 365 days, each with 24 hours. This results in the creation of yr:*year*, 365 *day* instances, and 365×24 instances of type *hour*. Each day is a component of yr, and each hour is a component of one of the days.

This, however only works for standard time units. If we wish to create a time period equal to an integral number of some standard time unit (*e.g.* 7 years) we use the ## notation in a slightly different way. For example: entering "*7yrs##(7,day)*" in the name slot, and *#year* in the type slot, results in the following:

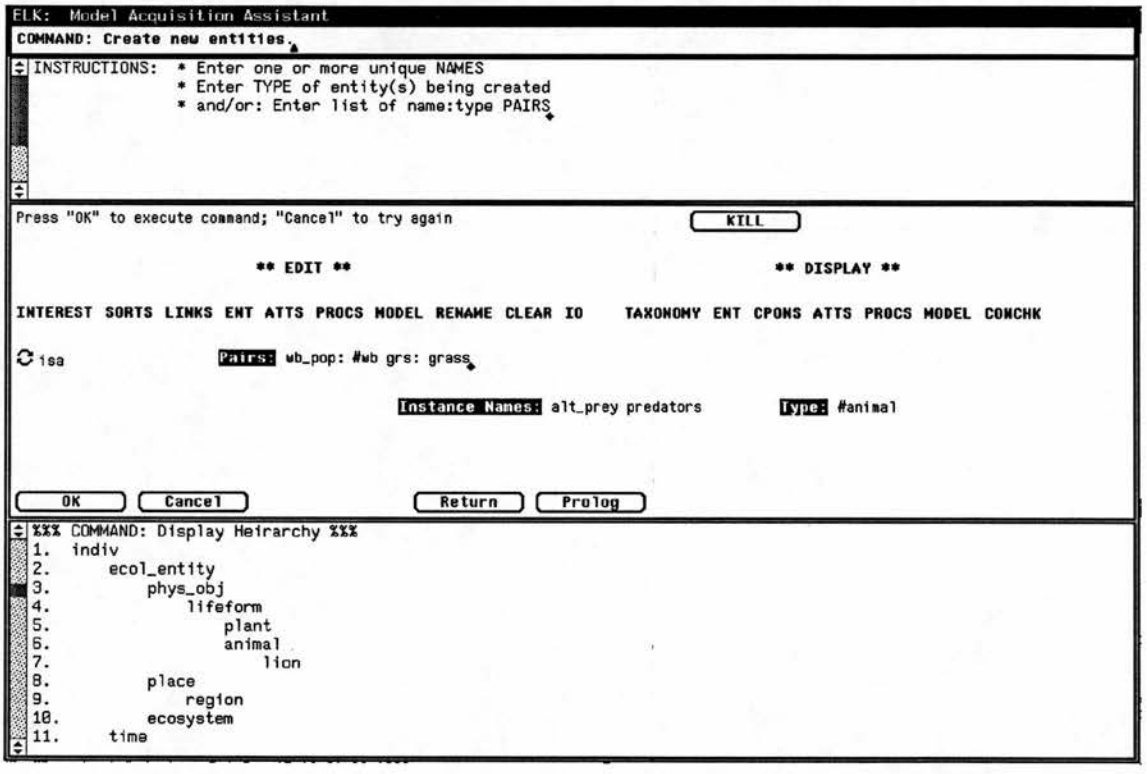
- Entities:
  - $7yrs$ is an entity of type #year
  - there are 7 entities of type *year*
  - there $7 * 365$ instances of type *day*
- Substructure:
  - Each year is a component of $7yrs$
  - Each year has 365 entities of type *day* as components

For example, $tim\_dim\_fn_2(7yrs, year, day)(2, 3) \subset tim\_dim\_fn_1(7yrs, year)(2)$.

If users wish to subdivide time entities in non-standard ways, they may do so by specifying substructure directly using the $\subset_\circ$ relation. This is discussed in § 7.4.2.

Consistency Checking

The system constrains the instances that may be created as noted at the end of § 5.3. For example, no entities are allowed whose most specific sort is *indiv*, sets



Users may define a number of entities simultaneously. Here, two animal collections, a wildebeest population, and a grass entity are created.

Figure 7–5: Creating Entities

are created only using #, never *set*. Also, an entity may not be an instance of two types unless one is a subtype of the other.

Transparency

There are various facilities for looking at the current set of entities. For example, by clicking the appropriate menu option, a user may list entities:

- in alphabetical order
- of certain sorts
- classified according to substructure

  e.g. only entities that are components of something

7.4.2 Substructure

There are three main ways to specify substructure. The most general one is to directly edit the component hierarchy in exactly the same way that the sort and part hierarchies are. The user selects the component taxonomy when selecting one of the commands for editing links. This defines the explicit pairs in $\subset_o$ from which $\subset$ is induced by transitivity.

The second way is via the special notation for specifying time substructure. This was discussed in § 7.4.1.

The other way is to use values of attributes to define substructure. This is one of only two significant interface challenges that we faced in designing ELK (the other is allowing users to build up complex nested expressions using the second order functions *average*, etc.). In all other cases, the interface design was fairly straightforward. The requirements for the attribute-based substructure specification interface are as follows:

1. Specification of very complex substructure should be possible with relatively few primitives.
2. The user should be able to define the substructure by interacting with a representation which is natural and easy to understand. In this case, an AND/OR tree is ideal.
3. The system should offer a menu of choices for attributes to be used as possible dimensions (*e.g.* *age*, *sex*).

4. The menu should be ordered so that the most likely to be used attributes are listed first.
5. The system should provides names for all the subdivisions automatically which the user may change as required.
6. The user should be able to view the substructure in the same way that they view the substructure defined by creating links in the component relation directly.
7. The system should be able to generate explanations of the ecological meaning of the entities.

The majority of this is implemented. The exceptions are 3 and 4. However, the utilities required for this exist and have been incorporated into the process interface; adding this to the substructure bit is a minor task. Another minor frill not yet included is allowing the user to give their own abbreviated name. For example, the system automatically generates the name *calf_wb_pop* in the example in § 6.8, but does not provide a way for the user to change it (*e.g.* to *wb_calves*).

The user begins by selecting the ENT–SUBSTRUCTURE–ATTRIBUTES-ADD command which indicates that they wish to edit/create an AND/OR tree specifying attribute-based substructure (see figure 7–6).[6] This may be used either to define an entity and its substructure from scratch, and/or to specify substructure for an existing entity. There are four slots.

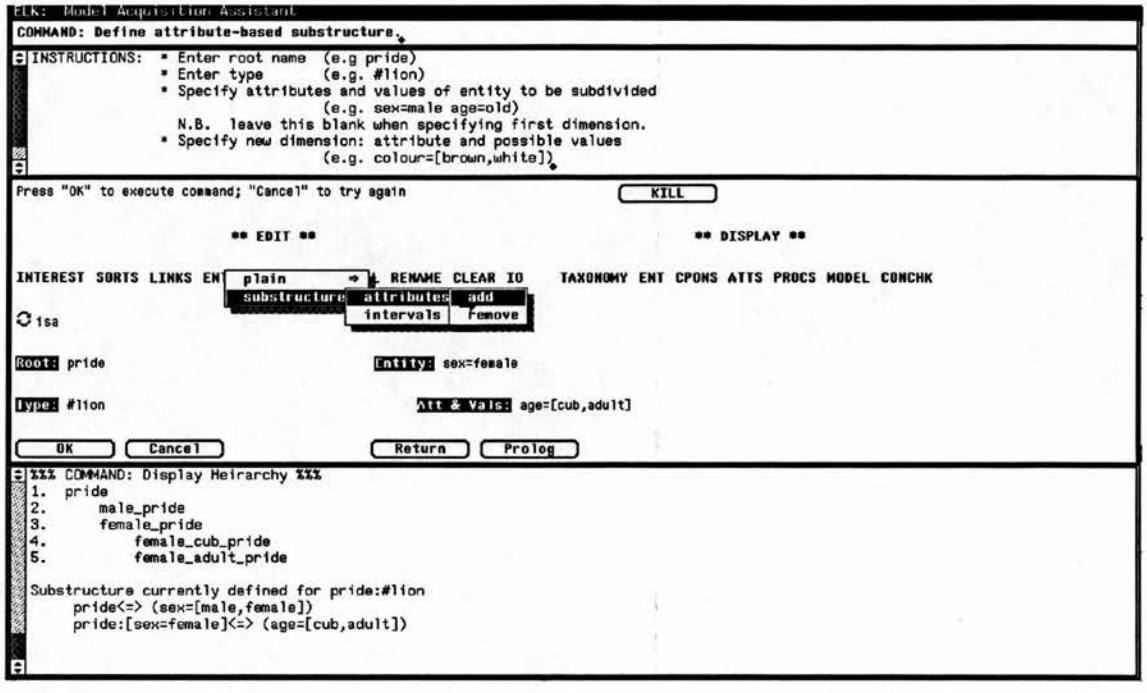
The first is referred to as the 'root'. This is the name of the entity to be subdivided (*e.g. pride*). When this slot is filled in, and if there is already some substructure specified for this entity, then the system automatically displays it; otherwise ELK prints a message indicating that no substructure is defined yet.

The second slot is the type of the root entity (*e.g. #lion*). This is filled in automatically if the entity already exists.

The third slot is for identifying the entity to be further subdivided. It is left blank when specifying the first level of the substructure; otherwise, it consists of a set of attribute-value pairs. For example, if the pride were already subdivided into male and female, and the female sub-pride were to be further subdivided, then the user would enter '$sex = female$' in that slot.

---

[6] The current version of ELK only handles pure or trees.



The user specifies that the male pride (not the female) is to be further subdivided by age. The current state of the substructure so far is given in both graphical and tabular formats. The graphical version is obtained by displaying the portion of the components hierarchy rooted at pride. Note that abbreviations are used, not complex logical terms. The tabular format is one way to represent the AND/OR *tree specifying the substructure.*

Figure 7–6: Attribute-Based Substructure

The final slot is for specifying a new dimension (*i.e.* attribute) which gives rise to additional substructure for the selected entity. This consists of giving an attribute name and a specification of a set of values for it (*e.g.* $age = [cub, adult]$).

The set of values may be given directly, or indirectly. For example, the default set of values is found in the dimension value space slot in the att_def construct for age of lions. This default may be selected by using '@$default$'. Users may also refer to sets such as $\{purple, black, brown\}$ by specifying $\backslash dark$ if each of those colours is a component of *dark*. The '\' notation is merely a shorthand for the user. It is used to select all the explicit members of a set.

Implicit Specification

ELK represents the substructure implicitly, so that if the set of dark colours, or the dimension value space changes, so does the specification of substructure. For

example, if something was subdivided by colour and the set of colours was specified as $\backslash dark$ then this gives rise to three subdivisions. If colours are added or removed from $dark$, then the number of subdivisions changes automatically. This gives a great deal of flexibility, although users must take care not to accidentally remove entities which may play a role in the model. The same applies when entities are explicitly removed.

Consistency Checking

No component-whole pair may be specified unless it is permitted by the possible component relation. This helps to ensure that only sensible substructure is defined. There are other things that should be checked for. These were discussed in § 5.6.5. For example, a branch should not be a component of two different trees. Currently we do not check the number of parts a composite has, or for overlapping which should be allowed only for homogeneous entities.

Relief from Redundant/Menial Tasks

The interface for defining attribute-based substructure is a powerful mechanism for defining complex substructure with the system doing most of the work.

Recovery Mechanisms

When component relationships are not permitted that should be, the system has the capacity in some cases to suggest a remedy. For example, suppose a user attempts to say some paw is a component of some lion, but $paw \not\subset^p lion$. If $paw \subset^p cat$, the system might suggest that one way to ensure that $paw \subset^p lion$ is if $lion \sqsubset^s cat$. The system could offer to make this change dynamically and continue the session normally. This is not implemented yet, but is one example of how we might use the type structure to provide recovery mechanisms which maintain the flow of dialogue.

Uniformity

When defining substructure using the component relation, *a user need not be concerned with the particular kind of substructure that they are defining.* By creating a link in the component hierarchy, they may be defining a member of a set, a

subdivision of a set, or a part of a composite. The system knows however, because of the types of the entities. It has to know, in order to permit it (via $\subset^p$).

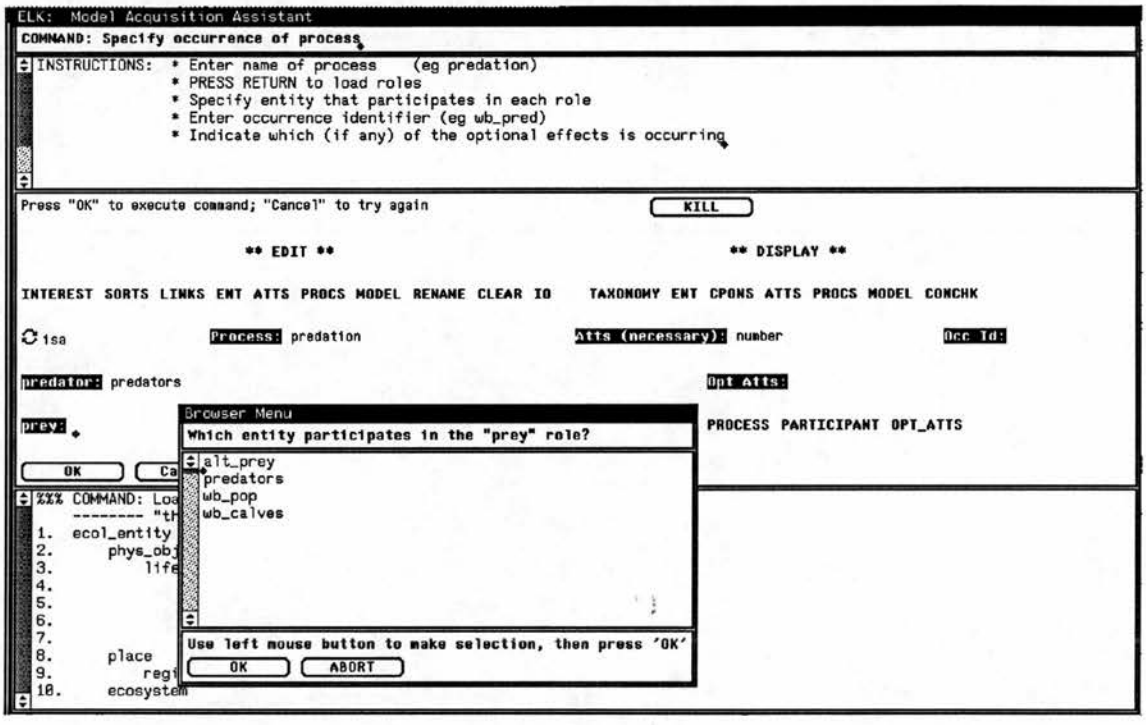
7.4.3 Attributes

The *att_esys* specifications are wholly implicit (see § 6.4.2.2). Thus there is nothing for the user to specify at the ecological system level with respect to attributes. However, users may query the ecological system and ask what attributes each entity has.

7.4.4 Processes

To have created the process *predation* at the general/ecological level is analogous to creating the sort *lion*. As there need not be entities of sort *lion* in the ecological system of interest, neither must there be any occurrences of predation. At the ecological system level, users say what processes are occurring between what entities, and what effects there are. In order to create an occurrence of a process, the process must be defined already. After selecting the edit command PROC–ECL SYS–OCCURRENCE–ADD, the user must choose the process that they are interested in. The choices are provided from a menu of processes including *predation*, *growth*, and any others specified at the general/ecological level. Once chosen, the labels for the role slots are automatically set (*e.g. predator* and *prey*. The users then enter the entities that are participating in this particular occurrence of the process (*e.g. predators*,*wb_pop*). Again, a menu of choices is provided which consists of all the entities of the sorts allowed by the *role_def* specification (*e.g. #animals*). Currently these are ordered in preference of entities whose types are closest to *#animal*. In the future, recency and interest may also be taken into account. Currently, the user must provide an identifier for this occurrence, although it would be easy for ELK to suggest a name (*e.g. grazing_1*). This results in the creation of *role_esys* specifications, one for each participant.

There are two slots for effects. One indicates what attributes are necessarily affected. This is filled in by ELK and may not be altered by the user; they are there as a reminder. The other is for the user to fill in. The system offers the menu of attributes which may be affected but not necessarily. The user picks 0 or more of these and enters them in the slot. This results in the creation of *effect_esys* specifications.



The user is specifying an occurrence of the predation process. For each slot, ELK *generates a list of choices often ordered in preference of those most likely to be chosen. This is a good example of how type checking can be used to simultaneously reduce search and ensure consistency.*

Figure 7–7: Specifying an Occurrence of a Process

Search and Consistency Checking

As for specifying processes at the general/ecological level, there is extensive assistance in reducing search by using type checking to generate menus. There is nothing new to say here.

7.5 The Simulation Model

Here we describe how a user may specify the simulation model. The simulation model further idealises the model encoded at the ecological system level which is itself an idealisation of the general/ecological knowledge base. We treat the dialogue and runnable-model levels separately.

7.5.1 Dialogue Level

Here we describe how a user may specify information *about* the simulation model that they wish to build, but which does not constitute part of the runnable model. There are three kinds of constructs: interest, goals, and user specified defaults.

7.5.1.1 Interest/Importance

By clicking on the INTEREST edit category, a menu pops up indicating a variety of things that a user may note interest in (as per § 6.5.2). Their specification entails filling the appropriate slots (up to 3) for identifying what it is that is of interest. They must also indicate whether for the purpose of this simulation model, this interest should be heeded or ignored. See figure 7–8 for an example.

Note that unless a user is prepared to directly input *qnam_tim* and *qnam_ent* expressions, this mechanism is not adequate for expressing interest in induced attributes. There is no obvious way that these may be listed along with the rest of the attributes in a menu to choose from. For a start, there are infinitely many. We can disallow nesting, but then we have to list every attribute several times, twice (*_tim* and *_ent*) for each qualifier. The most challenging interface problem we solved was with respect to eliciting complex nested expressions using the qualifiers *average*, *maximum*, etc. The requirements are:

- to allow arbitrary nesting of qualifiers
- to allow changing your mind

- to guarantee logically well typed descriptions
- to be understandable; *e.g.* by:
  - automatic creation of abbreviations (*e.g.* 'max_biomass')
  - automatic generation of English text to indicate the meaning of induced attributes nested 2 or 3 levels

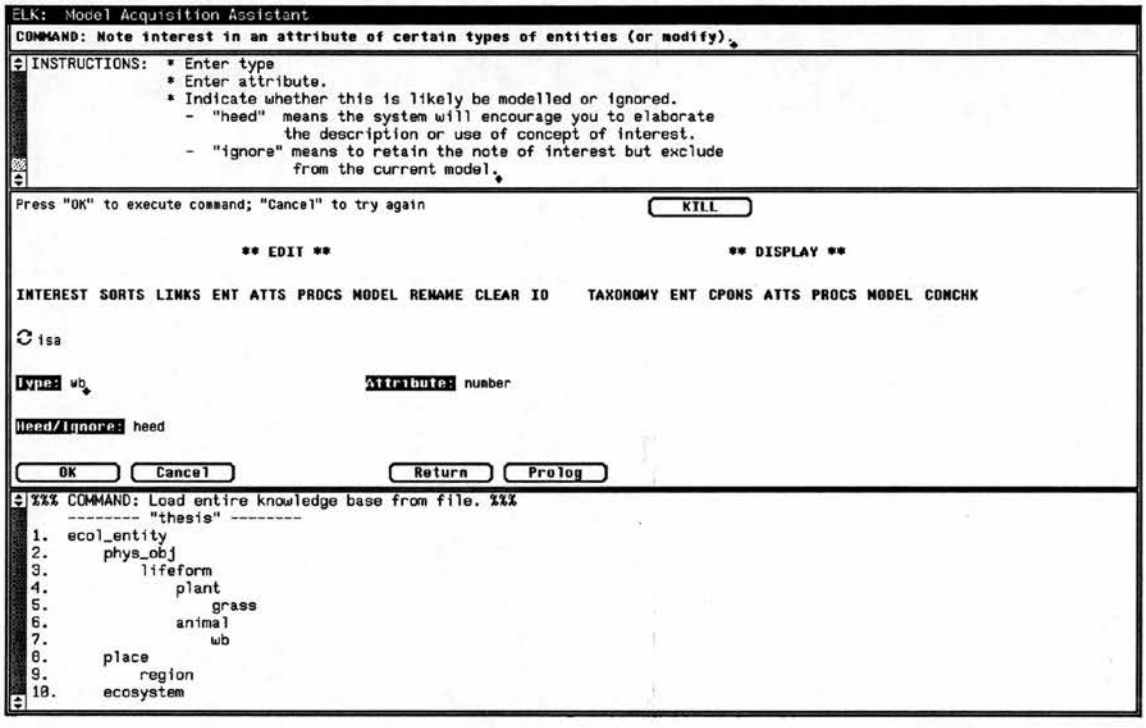
It is difficult to manually generate unambiguous text to describe the complex nested expressions. Yet ELK does this automatically. What made it possible to do this and to develop a usable interface was having the rigorously typed qualifiers; the *set* construct is fundamental. We describe this interface in the next section. N.B. the interface we implemented was developed in the context of the goal acquisition assistant, but it could be used equally well purely to note interest. As noted previously, the goal acquisition assistant is only partially integrated in the current version of ELK.

7.5.1.2 Goals

We have implemented a separate interface for eliciting goals of the form: "What is the affect of X on Y?". The result of defining such a goal is to instantiate an $xy\_affects$ construct. Currently we assume that X and Y are unary functions derived from ecological attributes of particular entities. We begin with the dependent variable, Y. Currently this subsystem is only partially integrated into ELK. An example showing the actual implementation is in figure 7–9. Below we describe a slightly enhanced version of this; which differs only in minor ways from the existing implementation.

A user begins by indicating the sort of entity that they are interested in, say wildebeest. The system then generates a menu of possible attributes to choose from, ordered in preference of those inherited by more specific sorts. At this point there is no need for a user to commit to whether they are interested in individual wildebeest or sets of them. So we also include in this menu *number* which applies only to sets of wildebeest. For reasons described above, we do not include the induced attributes using *average*, etc. in this menu. These are handled separately.

Suppose they choose *biomass*. If a user is interested in average, or maximum biomass, they select the QUALIFY option. This opens a separate window which enables a user to create complex attributes with arbitrarily deep nesting. They choose a qualifier from a menu (say average). The system offers the choice of what



The user specifies that for this simulation modelling exercise, the number of wildebeest is important. Different slots appear depending on what kind of thing the user is interested in. Only one slot would be necessary to express interest in the process of predation. The result of such specifications is for the system to give suggestions about what to do next with respect to the item of interest. The ignore option enables a modeller to record the fact that something is important, but that it is not included in the simulation model.

Figure 7–8: Noting Interest

the average is taken with respect to (*i.e.* time, or wildebeest). Time is always one option; the other option usually corresponds to the sort entered in the main goal window. If the attribute had been *number*, however, the other option would be the corresponding collection type (*e.g.* $\#wb$). This is because the attribute *number* does not apply to individual wildebeest. Thus it makes no sense to speak of the average number of a set of wildebeest unless we are really talking about a set of sets.

Suppose the user selects *time*. Formally, this means that we must λ-abstract on the 2nd argument of *biomass* to get the unary function required as input to *average*. The above interface events result in the new attribute:

$$qnam\_tim(average, biomass) =$$
$$\lambda W{:}wb.\lambda SetT{:}set(time).average(\lambda T{:}time.biomass(W,T), SetT)$$

ELK abbreviates this as 'avgT_biomass'; the 'T' denotes that the average is take with respect to time. More detailed translations are also possible. Users never see expressions containing $qnam\_$, λ, or even *set*.

Note that it is not necessary at this point to specify a particular wildebeest or a particular set of times. They may however specify the nature of the set of times (*e.g.* 7 days in a week). A separate window is used for this. With this modification, the above formal expression is changed by replacing $set(time)$ with $set(day)$.

In keeping with our design requirement to have a flexible system, the user is under no obligation to continue specifying the goal. The state of the goal so far indicates that the user is interested in how something affects the average biomass of wildebeest over some time period. If the user were to stop at this point, and never continue with this goal, it would have still served a useful purpose. It will have guided the user to thinking about what is important; the system can behave exactly as it would if the user had instead created the following interest specification:

$$att\_type\_interest(qnam\_tim(average, biomass), wb, manual, heed)$$

This interface, although described in the context of a goal acquisition assistant, could be used separately and explicitly for eliciting such interest specifications. The interest specification is used by the system to suggest how to continue the elicitation process as described in the previous section (*e.g.* to create model variables corresponding to this attribute).

If the user is interested in the maximum average biomass, they will want to continue. This is done by selecting the NEW option from the secondary goal window. This causes that window to be updated to correspond to the new level. The base attribute is now 'avgT_biomass' rather than biomass. The procedure is the same. The choices at this point are slightly different. The maximum may be with respect to time again, however at this level the option is not *time*, but *week* because of the specification at the previous level. The other choice remains the same; *i.e.* the maximum could be taken with respect to a set of wildebeest. Suppose the user chooses the latter option. The new attribute that now results is abbreviated as *max_avgT_biomass*; formally it is:

$$qnam\_ent(maximum, qnam\_tim(average, biomass)) :$$
$$set(wb) \times set(day) \mapsto positive$$

If the user was interested in the minimum over a set of sets of wildebeest of the attribute 'max_avgT_biomass', they proceed as before. The time option is still *week*, and the other option is now a collection of wildebeest, rather than individuals. The user may continue as long as they like, although in practice it is unlikely that nesting these qualifiers more than 2 or 3 levels will be required.

Suppose the user stopped at this point. What has been accomplished? The user has *not* *created* an attribute. The way *att_def* is defined, the nested *qnam* expression is already known to be an attribute. Neither has the user created entities, nor model variables. What they have done is express interest in a certain attribute for a certain type of entity. The *att_type_interest* construct will be created automatically, the effect being the same as if it had been directly specified. In this case, *auto* would replace *manual*. Fully specified goals with X and Y parts defined would give rise to more than one interest specification. The effect is to inform the system about the nature of the entities and model variables that are likely to be needed.

As a result of this partial goal specification, the system now knows that the user is interested in one or more wildebeest collections (not necessarily individuals), and in the maximum average biomass of these collections over one week intervals.

Through the process of gradual elaboration illustrated above, we are able to acquire quite complex descriptions from the user. The user starts with the simple general concept of a wildebeest. They then identify an important basic attribute. This is then qualified, gradually, one level at a time. Eventually the attribute

max_avgT_biomass is identified. This tells the system that the user is interested in wildebeest populations which they may be prompted to create as entities (*e.g. wb_pop*). They may then be prompted to create a model variable (creating an *att_var* specification). Lastly, the nature of the attribute is such as to require that the wildebeest population and time interval has specific substructure. The user may be invited to attach member wildebeest to the collection entity already defined. If the names of the individuals are unimportant, the indexing method may be used to define this substructure (*e.g. wbst(1), wbst(2)* etc.). The time substructure is defined similarly. For example, $thisweek{:}set(day), \forall i \in I_7.dy(i) \subset thisweek$ (*dy* is an abbreviation for the full *tim_dim_fn* expression). The variable created by this specification translates to the following expression in the object-level logic:

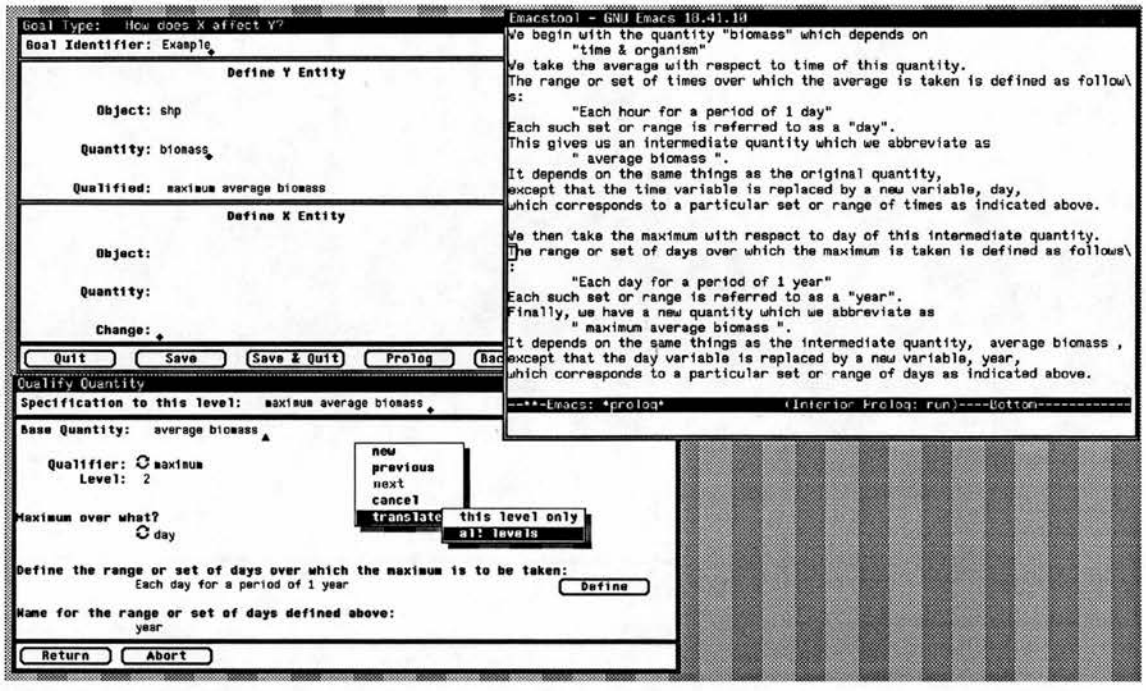
$$attvar\_fn(qnam\_ent(maximum, qnam\_tim(average, biomass)), wb\_pop) : time \mapsto positive$$

If this is to be computed in full, then *wb_pop* must have individual members (say $wbst(j)$ where $j \in I_n$. Also, variables must be defined (via *att_var*) for each of the wildebeest. The above expression evaluated at *thisweek* then expands to:

$$max_{j=1,n}\left(\frac{\sum_{i=1}^{7} attvar\_fn(biomass, wbst(j))(dy(i))}{n}\right)$$

Although we have discussed this interface in the context of goal elicitation, we see that goals are really just one possible way to begin thinking about the important things with respect to the modelling exercise. However, what we have described could be a stand-alone attribute/variable elicitation subsystem. As noted in chapter 3, getting the user to say what is important is one of the primary uses of goals. Additional uses for goals will be indentifying what the model output should be, and/or for defining simulation experiments.

There is a much quicker way to specify such complex nested variables for experienced users. Instead of going through the step by step approach, a user may input the abbreviation directly in any slot where an attribute is called for. For instance, if the user wishes to note interest in the average biomass of wildebeest populations over unspecified intervals of time, then they enter *#wb* in the type slot and *avgT_biomass* in the attribute slot for the INTEREST–ATTS–OF_TYPE–ADD edit command. If they wish to create a model variable corresponding to



The user is defining the dependent variable using the Define Y *window. They start by indicating what sort of entity they are concerned with. It is either typed in manually, or selected from the sort hierarchy in the main* ELK *window and loaded into the goal window. They then select the attribute of interest from a menu constrained to those appropriate to that sort. This is then qualified by average and maximum using a separate window. A complete, English translation of the state of the specification is automatically generated.*

Figure 7–9: ELK Goal Elicitation Interface

the maximum of such averages over different animal populations, then they enter *max_avgT_biomass* in the attribute slot when executing the ATTS–MODEL–DEFINE_VAR/PARM–ADD command. This is nearly equivalent to what was specified above; the difference is that no commitment is made to the nature of the time intervals.

These abbreviations are converted immediately to the internal representation using *qnam_* which is a more convenient notation for processing (*e.g.* type checking). Whenever they need to be seen by the user, the abbreviations are used.

Gradual Elaboration

We have illustrated the technique of gradual elaboration in three senses.

1. vague to precise
2. simple versus complex

   e.g. number of wildebeest, versus the maximum weekly average weight over a set of wildebeest.
3. ecological to simulation modelling concepts

   e.g. a wildebeest; a differential equation

We consider these in turn. Initially a user may express interest in wildebeest; by itself this is quite vague. A user could mean any number of things. This is made more precise by determining:

- whether it is individual wildebeest (and which ones) or groups of them (and which groups).
- what attributes of the wildebeest are relevant
- what other objects and/or via what processes affect the wildebeest
- etc.

The nesting of *maximum*, etc illustrates the second sense. This is made possible by defining *average* etc as higher-order functions rather than in the traditional way. The intimate association of these attributes with sets also gives ELK the potential to tease from the user whether they are interested in individuals or sets without their having explicitly to say so.

The third sense is the usual bridging of conceptual distance that we discussed in chapter 6, and again here. This is facilitated primarily by the distinctions embodied in our knowledge ontology.

Transparency

This is demonstrated by the ability of ELK to produce coherent English text explaining the meaning in ecological terms of the complex attributes (see figure 7–9).

7.5.1.3 User Specified Defaults

The *att_inh* and *effect_inh* constructs are instantiated using an interface which is quite similar to that for the corresponding *_def* constructs. We do not elaborate here. An example defining an *att_inh* specification is given in figure 7–2. One point to note is the fact that when the slots correspond, the default values for these defaults are loaded directly from the appropriate *_def* specifications. Thus,

if the effect really is a transfer of materials, then by default it will be modelled that way. The value spaces for attributes remain the same, etc. The *_def* specification cannot supply the *var_spec* slot in the *att_inh* construct because variables are a modelling concept; the *_def* constructs specify only general/ecological information. However, if the attribute is a constant, then *parameter* is the default setting. Otherwise, we arbitrarily let *stvar* be the default choice.

Relief from Redundant/Menial Tasks

This is the main reason for the user-specified defaults. These are but one example in the wider context of defaults that the system provides for the convenience of the user. Even if there are no explicit *att_inh* constructs, the system has its own defaults which tend to reduce the amount of explicit specification required from the user. They are acquired mostly by using the information in the general/ecological knowledge base.

7.5.2 Runnable Model

The two main aspects of specifying the runnable model are:

- creating model variables
- specifying the computation network via schema instantiation

These determine the differential and non-differential equations respectively. This is where many important idealisation decisions are made. Users decide which aspects of the ecological system need to be included in the simulation model. For our simple Serengeti example, there are just a few entities, each having several attributes; and there are a few processes, each having several effects. This amounts to on the order of several dozen possible variables that could be created. Yet the runnable model that we require uses just over a dozen. For those aspects that are included, users must determine how they are to be idealised.

For example, the number of predators (*n_pred*) is held constant, but the number of wildebeest (*n_wb*) is a state variable. Choices also exist for modelling effects. Even though in the real world the effect of predation is to transfer biomass from the prey to the predator populations a user might choose only to model the effect on the predators and ignore the effect on the prey. The possibilities are numerous. Most effects give rise to partial rate variables for which some way of determining

its value at each time step must be specified. This is done by selecting equations or instantiating schemata.

7.5.2.1 Model Variables

The interface for defining ecological model variables is similar to the others. There is a slot for each argument in the *_var* constructs. For attribute variables, there are two idealisation decisions. One is about the value space, which will usually be unchanged from the *_def* specification. The other is the type of variable. This is an important decision.

There are also two idealisation decisions for effect variables. One is the time scale, the other is whether the effect variable will be a partial rate variable or not.

Pure model variables are created using a virtually identical interface as the ecological variables. The only difference is the lack of ecological information. Thus for pure partial rate variables, the user has to explicitly state what the affected variable(s) is(are) and whether they are incremented or decremented (*transfer*, *inc/decrease* etc are not required). As noted in § 6.4.3.2, these are automatically filled in for effect variables.

Unless interest has been noted (possibly via a goal), the menu for attributes that is generated when a user is creating a model variable will exclude induced attributes (*i.e.* using *qnam_* constructs).

Consistency Checking / Recovery Mechanisms

There is a large amount of consistency checking that goes on here. For example, when defining effect variables, the idealisation choice cannot be *dec/increase* if it is *inc/decrease* in the *effect_def* specification. In all cases, the affected attribute must have a corresponding *att_var* specification to enable equations to be put together properly. This is another situation where a recovery mechanism could be installed. It would be a simple matter for the system to offer to create the variables automatically. If the idealisation of this effect variable is one of *increase*, *decrease*, or *transfer* the variable must be a state variable; if it is *change*, then the idealisation of the corresponding attribute variable is set to *intvar*. If the idealisation for an effect variable is *change*, then the corresponding attribute variable *must not* be a state variable.

Another check must be made when the *transfer* option is used. Specifically, the value spaces for the two corresponding attribute variables must be compatible. For example, it cannot be that the idealised value space for one is real numbers, and the other $\{large, small\}$. Finally, the time scales must be ecologically consistent. Except for the latter two (and the recovery mechanism), all of the above consistency checking is implemented. Messages are printed indicating the nature of the problems, and where possible how to fix them.

Implicit Specification

The users specify only the variables and their types. This implicitly defines the set of differential equations.

Transparency

The explanation of the ecological variables in ecological terms is one of the major ways of facilitating model comprehension (see examples in § 6.8).

7.5.2.2 Computation Network

The MODEL–EQUATION command is used to specify the equations and schemata that comprise the computation network. The user first specifies which variable they wish to specify how to compute. For this, ELK provides a menu containing all the proper variables (*i.e.* excluding parameters). The exclusion is because it makes no sense to specify how to compute a parameter. It is easy to change a parameter into a variable (and vice versa) after which it would then show up in the menu. The proper variables in the menu are divided into two groups, based on whether an equation has already been specified for the variable. These lists are each in alphabetical order except that the partial rate variables (*i.e.* the root nodes to the computation network) come first. This is because we suspect that users may wish to specify these first, although they are free to do things in any order. Suppose the variable *wb_eaten* is chosen. The user has two options at this point. They may select from the list of pure schemata, or from the list of appropriate ecological schemata. In this case, a simple pure schema is appropriate. *wb_eaten* may be computed using the pure schema *dirp* for direct proportionality. The generic equation ($f = x * a$) is displayed in the equation slot, and the user must decide which variable gets attached to each input. For each input, ELK

generates a menu of variables to choose from. When all inputs are specified, the user presses OK and the schema is instantiated.

For ecological schemata, ELK offers rather more assistance (as noted in § 6.4.4). Here we describe a complex ecological schema and the support that ELK provides. Recall the equation for computing the specific rate of predation (2.7, page 44). There are a total of six inputs to this equation, however this would vary according to the number of competing prey populations. This equation is a special case of a more general relationship. In particular, the denominator is a summation of the product of three attributes for each of the competing prey populations. The three attributes are average handling time, average capture coefficient, and population density. An abbreviated version of the general equation, as stored in the ecological schema in ELK is given below. Recall that $avg\_htime(Y,T)$ is an abbreviation for $average(\lambda X.htime(X,T),Y)$.

$$spr\_pred(\$predator)(\$prey,T) = \frac{avg\_cap\_cf(\$prey,T) * pop\_density(\$prey,T)}{1 + total(\ \lambda Y.(avg\_htime(Y,T) * avg\_cap\_cf(Y,T) * pop\_density(Y,T)),\ \{\$prey\} \cup \$c\_prey\ )}$$

Recall that the variables prefixed with a '$' are dummy variables that are matched before a schema is instantiated. Suppose the user decides to specify an ecological schema for the variable *spr_pred_wb*. Because of the explicit association of this variable with the attribute *spr_pred(predators)*, $*prey* matches with *wb_pop*, and $*predator* matches with *predators*. The match only succeeds if the precondition *predation(predators, wb_pop)* holds; it does in this case. All schemata which match are offered in a menu. Currently, although all the machinery is generally applicable, ELK had only two exemplar schemata in its knowledge base. Thus only one ecological schema is in the menu for *spr_pred_wb*; users may of course choose any pure schema instead.

$*c_prey* is the set of competing prey populations (excluding $*prey*). ELK infers this from information specified at the ecological system level. ELK determines all the occurrences of the process of predation for which *predators* is in the predator role. It then collects all the entities in the *prey* role for each of these and removes *wb_pop*. As for any other part of the ecological system specification, the user is free to ignore some of these. ELK provides the list of possibilities, and the user chooses the ones they wish to take into account.

This is all that the user needs to do (see figure 7–10). ELK makes the appropriate connections updating the computation network as required (see figure 6–1). ELK makes the appropriate substitutions for the dummy variables; the attributes are replaced by the corresponding model variables. For example, the attribute $qnam\_ent(average, cap\_cf)$ of the wildebeest and alternate prey populations is replaced by wb_cap_cf and ap_cap_cf respectively. Note that the number of variables depends on the number of competing predator populations.

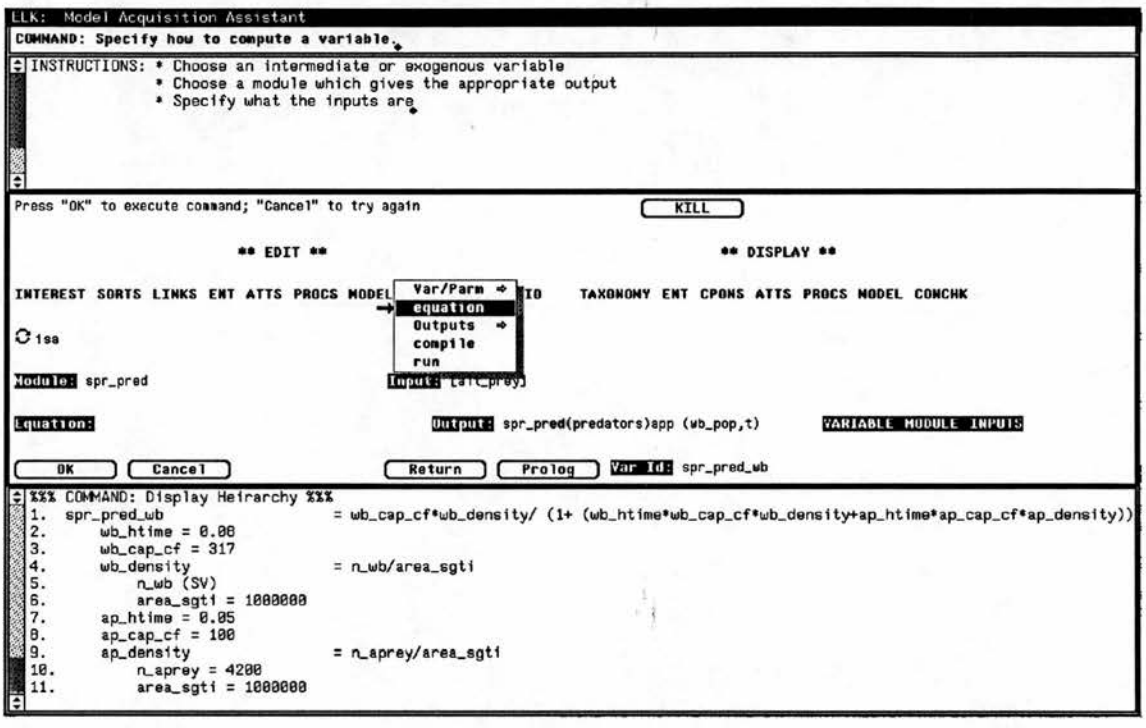
This example shows how even though the predation involving the alternate prey is not explicity modelled, it's presence in the ecological system level is used to assist in constructing parts of the model which assume its existence.

Also, the equation is expanded getting rid of λ's and $qnam\_$ expressions where appropriate. In this case, the term with *total* is replaced with an explicit sum (see figure 6–2). However, the two uses of *average* are not expanded. This is for two reasons. First, the terms using *average* correspond to parameters, not proper model variables; they are not computed. Second, even if they were proper model variables, there are no explicit members over which to perform the computation.

This illustrates an important point about the use of the higher-order functions average, maximum, etc. In particular, *terms using these higher-order functions need not be expanded to enable computations over explicit sets.* Whether or not such expansion/computation takes place is the user's choice. It is unlikely that something like average handling time would ever be computed directly. However, it might be possible to estimate it from other field data; this would mean a separate equation would be used. Thus, *just as for any other variable*, the user is free to select the equation to compute it, or to let it be a parameter (*i.e.* constant). Using these higher order functions this way still enables explaining the meaning of the variables corresponding to these induced attributes (*e.g. wb_htime*). Furthermore, ELK is flexible enough to allow the user to change their mind and decide to compute some average (or maximum, etc) directly. The meaning of the variable would not change, only the way it idealised the attribute of the entity it represents.

Consistency Checking / Search Control

Using type information to match ecological schemata greatly reduces the search space for selecting equations. Other menus are provided as noted above.



The user selects a schema for computing the specific rate of predation of predators on the wildebeest population using the MODEL–EQUATION *command. The single input in this case is a list of prey populations other than* wb_pop. *The display in the lower portion shows the instantiation of the general equation for computing specific rate of predation given any number of competing prey populations. The inputs are presented in graphical format enabling users quickly to see what the variable dependencies are. State variables are denoted by (SV), and values for parameters are shown.*

Figure 7–10: Instantiating Ecological Schema

Relief from Redundant/Menial Tasks

As illustrated by this example, the degree of assistance offered by ELK for ecological schemata can be quite high. The savings are less dramatic, but still worthwhile for simpler schemata like pop_density (see § 6.4.4).

Transparency

After a schema is instantiated, the updated computation network (with or without equations) may be displayed (see figures 6–1 and 6–2). This gives users a concise view of the dependency relationships between the different variables and parameters. Although not currently implemented, text may be generated for each instantiated schema explaining what the relationship is that is being used to compute each variable. More information is available for the ecological schemata.

Uniformity

Both pure and ecological schemata are specified using the same basic command. The listing of the computation network uses the same display command as for all the other hierarchies.

7.5.2.3 Running the Model

Before a simulation may be run, users must specify the simulation time and model output. The former is a special entity called *simtime*; it is created just like any other time entity with substructure (see § 7.4.1). For example, if we wish to run the Serengeti model for a 20 year period, then we can specify *simtime*##(20) in the name slot and *#year* in the type slot using the command for creating entities.

To specify the model output, a user must say which variables are to be printed at what times. Currently, the facilities for this are quite primitive. Users may only provide a list of model variables. ELK assumes that these are to be output at every time interval. A useful extension would be to allow the specification of selected times (*e.g.* every other year). Another extension would be to compute average, maximum, etc over specified sets of times. For example, a user might be interested in the average number of wildebeest over all or part of the simulation time period. With *simtime* specified as a set of 20 years, a user should be able to specify $avgT\_number(wb\_pop, simtime)$ as an output variable with-

out having to specify an equation for computing it. The system should then store up the intermediate values as required during the simulation and perform the computation at the end. It should also be possible to compute the average for the first and second 10 years of the simulation. This could be done by instead specifying $simtime\#\#(2, decade)$ and then specifying the output variables as: $avgT\_number(wb\_pop, decade\_simtime(I))$ for I equal to 1 and 2. N.B. $decade\_simtime$ is the abbreviation for $tim\_dim\_fn(simtime, decade)$ whose type is $natural \mapsto decade$.

In the above expression, $decade\_simtime(I)$ may be viewed as an instant representing a whole decade. However, if we are directly to compute the average number of the wb_pop for each year of this decade, it must then be viewed as a set of 10 years. This is fine, since there are 10 components of type year automatically created. In running the simulation, ELK must know when to compute the average. To do this, ELK has a notion of time equivalence which identifies $tim\_dim\_fn(simtime, decade)(I)$ with $tim\_dim\_fn(simtime, decade, year)(I, 10)$. So when the simulation gets to this year, the computation may take place. This notion of time equivalence is implemented, but not the ability to perform these computations over sets of times during a simulation.

If any other time entities were created and used, they must map correctly to the simulation time line. No such facility is currently provided, although it should be conceptually straightforward to do so by providing a command for stating equivalences between time entities.

The commands for compiling and running the model are quite trivial, we omit any details. The actual output provided by ELK for the example model is given in figure 7–11.

Overall, the current facilities for running models with ELK are very crude. The point of this exercise is to demonstrate that the constructs provided at the runnable-model level are sufficient to specify models that really run.

```
year 1
    n_wb = 11032.0
    spr_pred_wb = 2.48705
year 2
    n_wb = 12236.7
    spr_pred_wb = 2.68851
year 3
    n_wb = 13655.1
    spr_pred_wb = 2.91367
```

This shows the model output for three iterations of the example Serengeti model. There are two output variables. Note that each iteration corresponds to meaningful time steps which correspond to the specific structure specified by the user for the overall simulation time.

Figure 7–11: Model Output

7.6 Testing Elk

In addition to various toy examples, Elk has been tried on various models published in the ecological modelling literature. Initial results are encouraging. We have implemented the Serengeti model completely, using ELK to define from scratch the sorts, functions, processes for the general knowledge base as well as to define the specific model variables, parameters, equations, etc that comprise the model, right through to running the model producing output. All parts of the general/ecological knowledge base may be created and modified by the users except for the ecological schemata. Various other examples have been partially encoded in Elk.

Much of the current version of ELK has been very recently implemented, and thus has not been tested on users other than the author. Earlier versions were tested with about a dozen people. Again, results were encouraging. The goal elicitation subsystem was tested separately. Users found that the template "How does X affect Y" was a useful one capable of capturing a good portion of the goals that they had in mind. The major usefulness of these early tests was to point out limitations in the expressive power required to capture the enormous variety of concepts that X and Y can be. A great many of these problems have been solved.

Several users tested an early version of ELK that was limited to describing the sort and part hierarchies as well as creating entities and component hierarchies. The interface as described in this chapter was used. It was necessary to

spend 15-30 minutes explaining the basic ideas and distinctions that they are required to make. Chief among these is understanding the difference between the sort and part hierarchies, and the difference between the possible part hierarchy (*e.g.* $paw \prec^p lion$) and the component hierarchy (*e.g.* $paw1 \subset lion1$). These users came along with their own problems which were in no way suggested by the author. Initially, I acted as a knowledge engineer asking them various questions and modifying the knowledge base as I saw fit. A number of interesting representational problems were uncovered; these have been discussed in chapter 5. For example, the homogeneity distinction.

In some cases, users were left on their own. This worked out fairly well as long as I was around to answer a few questions now and then.

7.7 Design Rationale Revisited

In chapter 4 we described the design rationale for ELK. The major requirements are: model comprehension, expressive power, conceptual distance, search space control, consistency checking, flexibility, and relief from redundant/menial tasks. In chapters 5 and 6 we showed how we help meet the first three requirements all of which are directly related to the formalism. In this chapter we have concentrated on the latter ones all of which are directly concerned with the provision of assistance via a friendly interface. We have also elaborated on the issues of conceptual distance and model comprehension. Chapter 6 showed how the formalism supports bridging conceptual distance, and facilitates model comprehension. In this chapter we saw how this is manifest in the interface. In this section we summarise the key techniques used to help meet these major requirements.

7.7.1 Transparency / Model Comprehension

As originally conceived, the term 'model comprehension' referred to the ability of a user to understand the simulation model in terms of the ecological system being simulated. However, if we use the more general meaning of the term 'model', this refers to the ability of users to understand the general/ecological knowledge base and the description of the ecological system as well. All of this is part of the general requirement of achieving transparency in the interface.

An important design feature of the ELK interface which facilitates transparency is *uniformity.* We have attempted to use similar techniques for accomplishing similar tasks wherever possible. Some examples of this:

- The same commands are used for editing and displaying each hierarchy.
- All attribute and variable related interface commands are similar across all levels.

Uniformity in the interface is facilitated most directly by uniformity in the underlying formalism. We use similar constructs to represent similar concepts. For example, there are various hierarchies all represented the same way; we have used naming conventions as well in a consistent uniform manner (*e.g.* *_def*, *_inh*, and *_var*).

ELK currently provides extensive facilities for examining the state of the specification (see § 6.8). Many more are possible. The interface would be greatly enhanced by using graphic display of the hierarchies. Although very important for an end product, their incorporation is more development than research. ELK demonstrates that in principle a friendly user interface is possible to construct.

7.7.2 Conceptual Distance

We covered this in some detail in § 6.9; here we review the main points. The primary technique for reducing conceptual distance is by constructing a bridge linking the ecological concepts to simulation modelling ones. This is manifest in the knowledge ontology which gives rise to three distinct idealisation phases:

1. construction of general/ecological knowledge base
2. description of ecological system to be simulated
3. specification of runnable simulation model

The third is the most important and most difficult. The dialogue level specifications assist in bridging levels 2 and 3.

Crucially, each of the four levels comprising our ontology consists of constructs whose semantics are expressed in terms that are readily understood by ecologists familiar with basic modelling concepts. This facilitates the development of:

- friendly interface facilities because there can be a direct mapping between the constructs and the interface actions.

- translation routines which are used to achieve transparency in the interface, which is important for model comprehension.

7.7.3 Search Space Control

In chapter 1 we identified two kinds of choice that need to be made (*what* to do and *how* to do it) and three levels of assistance in search (*identification*, *pruning*, and *advising*).

What to Do

Choosing *what* to do is a control issue. In ELK, this is manifest in deciding how to proceed in the elicitation process (*i.e.* what command to execute). There is also a secondary level of what choices to do with which slot to fill in first; this is sometimes important. The *identification* of this search space is implicit in the set of available commands, which is in turn derived from the available constructs in the formalism.

The *pruning* of this search space is manifest in the fact that certain specifications are required before others as well as certain slots needing to be filled in before others for a single command. For example, you cannot create *grs_wt* as a model variable representing the weight of the grass unless the attribute *weight* exists as well as some entity of sort *grass* (say *grs*). This is not an isolated example: there are a wide variety of such restrictions. For a specific concept, the user is forced to specify the general/ecological knowledge base first, then the ecological system, and finally the simulation model. This bridging of the conceptual gap is a direct consequence of the knowledge ontology.

We currently provide no on-line *advice* on what is the best command to execute next, given various sensible options. However we do for each command give explicit instructions on the order in which slots should be filled in. We have identified a small number of general guidelines, such as it being a good idea to elicit goals first. These will be included in any documentation for users. Much of the time, it makes no difference what order things are done in (*e.g.* which parameters to create first).

The provision of extensive useful advice on what to do next (including identifying when it does not matter) requires a major knowledge acquisition effort. Many modellers should be consulted, many strategies should be empirically tested. The

result of this would be a body of mostly heuristic knowledge which characterises one or more good control strategies for constructing models. Once identified, this knowledge could be put to use (possibly as a separate rule-base subsystem) to control an agenda, or suggestion box.

How to Do It

The *how* choice is manifest in two ways in ELK depending on the nature of what it is that needs to be done. We shall refer to what needs doing as a modelling subgoal because it constitutes some task that must be achieved in the modelling process. The simpler of the two is deciding how each construct is instantiated, more specifically, what the entry for each slot may be. There is also a higher level *how* issue regarding possible ways to achieve some modelling (sub)goal which itself may be a high-level notion not directly expressible in our language. A good example of this is how to represent a concept (*e.g.* black sheep as discussed in § 4.4.2). In this case, deciding how to do something may consist of using one construct or series of constructs to achieve the subgoal instead of others. We refer to these two kinds of 'how' choice as low- and high-level.

In ELK, the *identification* of the search space for low-level how choices is achieved by presenting the available choices for each slot. Mostly, this derives from the typing. For instance, when a variable corresponding to an ecological effect is being created, there are explicit idealisation options (*e.g.* the instances of the meta type *chg_spec*). We do not identify except implicitly the choices for high-level how decisions. This would require another meta level in which one could express notions like "representing a concept".

We *prune* the low-level how search space by using general/ecological and modelling knowledge. For instance the option *increase* is not allowed for idealising the effect of predation on the biomass attribute of the prey. We do no direct pruning of the high-level how search space. We have made no real attempt to characterise it yet. This is an important topic for future work which is discussed briefly in chapter 9.

We offer no on-line *advice* on how best to instantiate constructs, nor on how best to achieve high-level modelling subgoals (although we do at times order menus such that the options most likely to be chosen are listed first). These are difficult decisions that lie at the heart of any modelling problem. There are important

choices to be made at each level (in the ontology), and between levels. For example, we saw that there were various ways that the concept of black sheep might be represented ranging from virtually not at all in *black_shp:animal*, to the fairly rich version: *black_shp:sheep(black)*. None of these choices is right for all problems; but each would be appropriate for some problems. The philosophy in designing the formalism for ELK is to make it as expressive as possible to give users the option to represent things in a rich and general way when it is useful to do so. We do not advocate using all the expressive power all of the time. We offer the following general rule for deciding how to represent a given concept:

> *The more likely a concept is to be used frequently, the richer and more general its representation should be.*

Although important, we make no claims about this being original or interesting in its own right. It is good knowledge representation practice.

At the simulation modelling level, the *how* decisions are the usual ones that ecological modellers make. Should something be a parameter, or a state variable? How should an intermediate or exogenous variable be computed? Should an aggregated representation be used? In the example model, *predators* is an aggregated population, the subdivisions are explicit. The decision to use the aggregated version is made at the simulation level. It is manifest in whether variables are defined for the lion and hyena sub-populations or just the predator population. These are difficult choices at the heart of any modelling problem. Although some general principles can be identified, the knowledge required to provide useful advice for these choices is not widely available.

Summary: Search Space Control

By far the most support we offer is in identifying and pruning the various idealisation search spaces. In the context of an ecological modelling assistant, identification of the search space constitutes a major achievement. Without ELK, virtually all the 'how' choices are high-level ones. With ELK, a user may construct models slowly and gradually being led along the way. Pruning is also a major benefit; it is a consequence of the consistency checking, which in turn depends heavily on the rich type structure and the four-level knowledge ontology embedded in the formalism. Summarising:

- The language for representing general/ecological knowledge (*i.e.* PESAV framework) identifies the search space for describing ecological domains. This framework also embodies a small amount of pruning.
- general/ecological knowledge base identifies and prunes search space for describing ecological systems.
- the language for representing simulation models in conjunction with the ecological system description, and the mapping between ecological concepts and modelling ones identifies and prunes the simulation modelling search space.

Provision of advice both in controlling the order in which models are specified, and in making good modelling decisions is largely absent in ELK. This is where goals may re-enter the scene. What constitutes the best way to model something depends on many things, but perhaps most fundamentally on the goals of the modelling exercise. The knowledge required to make use of goals is not readily available; getting it constitutes a major knowledge acquisition exercise.

7.7.4 Consistency Checking

We have seen many specific examples of consistency checking. A fundamental aspect of ELK is that consistency checking is used to prune the idealisation search spaces. Here we briefly review the consistency checking in ELK both in terms of what is provided and what techniques are used. We ensure consistency:

- in the general/ecological knowledge base with respect to the world by constraining it to fit into the PESAV framework.
- in the description of the ecological system with respect to the general/ecological knowledge base.
- in the simulation model by with respect to the ecological system description in conjunction with basic modelling knowledge.

There are two modes of consistency checking:

- *a priori* checking: dynamically generated menus which offer only sensible choices
- *post priori* checking: validating inputs, issuing warnings as appropriate

Primarily, the support for consistency checking derives from:

- Explicit separation of the general/ecological knowledge base from the ecological system description enables ensuring consistency of the latter.
- Explicit separation of the description of the two ecological levels from the simulation model enables ensuring consistency of the latter.
- Use of typing (object- and meta-level) is at the heart of virtually all consistency checking in ELK. A great deal of semantic consistency is ensured using purely syntactic techniques.

7.7.5 Flexibility

We distinguished two main kinds of flexibility which were major design constraints:

- *control*: can do tasks in any order
- *modifiability*: easy to modify/remove existing specifications.

We require the former because we believe every ecologist will have their own preferences that on the whole should be catered for. Thus in ELK, users can mostly do whatever they like whenever they like. This is not always possible, of course. For example, a user may not create an attribute variable unless the attribute and entity already exist in the general/ecological knowledge base and ecological system description respectively. Thus, *for a particular concept* we force users to go from the general/ecological level to the runnable model via the ecological system level. However, they may freely choose whether to describe the bulk of the ecological system first, or to interleave this with creating model variables. They are under no obligation to specify anything at the dialogue level. This is for their convenience only. While we think that specifying goals is a good idea, we do not force it.

We require modifiability because the process of modelling is difficult and requires a lot of experimentation, trying different ideas etc. To achieve this we have for every construct provided commands for modifying and/or removing existing instantiations. We provide ample messages and warnings indicating what the consequences of a users action are. The potential for changes causing related specifications to be updated is great. We attempt to identify where changes cause immediate problems of consistency, however we make no claims that we catch everything. One important area for future work would be to try to come up with a general consistency checking methodology which would replace the current ap-

proach which largely relies on the system designer identifying all possible problems beforehand.

One step in this direction is extensive use of *implicit specification* which we already have in ELK. In a great many cases, when specifications are added, removed, or modified, the related specifications are automatically updated. One example is the differential equations; see § 6.6 for others.

This flexibility is traded off against computational speed. On the whole, this has not been a problem for ELK because at any one time, there is not much computation to be done.

7.7.6 Relief from Redundant/Menial Tasks

We have provided a wide variety of mechanisms for relieving the user of tedium of one kind or the other. These are classified as follows:

- implicit specification
- defaults
- recovery mechanisms
- miscellaneous

Implicit specification, besides facilitating modifiability, also saves the user of a great deal of work by reducing the amount of explicit specification required.

ELK has an extensive capacity to supply default values. This is particularly evident in the *_def*, *_inh*, *_var* sequence of meta-level constructs. When creating variables, the default values are always taken from the corresponding *_inh* specification. Usually, these are implicitly defined, the [default] defaults coming from the corresponding *_def* construct. These are guesses that ELK makes about what the most likely choice will be. Users may override these with explicit *_inh* specifications as required.

In principle, ELK has an extensive capacity to provide recovery mechanisms so that users when they make mistakes can be relieved of the tedium of manually making fixes. We have identified a number of situations where these would be useful; a few have been implemented but not yet been incorporated in the current interface. For example, if a sort is not yet specified, the system could offer to present the link command for editing the sort hierarchy and then go back to where they left off. Currently, a user must go through the bother of cancelling

the current command, selecting the command which will make the appropriate fix, and then remembering what it was they were trying to do in the first place.

Collectively, these recovery mechanisms are more than mere bells and whistles. Extensive use of ELK would no doubt result in the identification of many other situations where recovery mechanisms could be installed that we have not mentioned yet. The typing, which is the foundation for consistency checking is what makes it possible to provide this feature. Taken in conjunction with a wide variety of other useful aids, a suite of recovery mechanisms could greatly enhance the usability of ELK.

The automatic generation of menus constraining choices is another useful aid. When the menus are ordered, this constitutes another case when the system is providing defaults.

7.8 Conclusion

In chapters 5 and 6 we demonstrated that we have a significant amount of expressive power. In this chapter, we described the details of the user interface. We began by giving a general overview of what facilities ELK has and how it is used. We then gave examples of how the system acquires formal descriptions of each important kind of information in our ontology. These are:

- general/ecological knowledge
- the ecological system of interest
- modelling goals
- what the user is interested in
- user-specified defaults
- a simulation model of the ecological system.

For each, we illustrated the role that the information plays in meeting one or more of the major design requirements. Finally, we summarised the key requirements and the techniques used to meet them. Of particular importance is the issue of identifying and controlling choices which constitute the idealisation search spaces. The three phase approach is fundamental to the system's ability to to manage choices.

Conclusion: Part II

This concludes the second part of the thesis. The first part consisted of

1. defining the general problem of formalisation
2. exploring the particular formalisation problem of ecological modelling
3. designing a computer assistant to alleviate the ecological modelling formalisation problem

In the process, we identified one major objective and two major hypotheses. The objective is 'to identify how goals may be used to assist in the ecological modelling formalisation process'. This was addressed in chapter 3. The two hypotheses are:

1. *ontology completeness hypothesis:* That every piece of information that is deemed to be useful in the process of constructing ecological models can be unambiguously placed into one of the following categories:
   - general/ecological knowledge
   - ecological system description
   - dialogue control information
   - runnable simulation model
2. *ontology usefulness hypothesis:* That this knowledge ontology is instrumental in helping achieve the following benefits in the context of a computer assistant for ecological modelling:
   - model comprehension
   - sufficient expressive power
   - small conceptual distance
   - identification and control of the idealisation search space
   - ensuring consistency

The second part consisted of:

- describing the solution to the ecological modelling formalisation problem.
  This was achieved by:
- describing the theory and implementation of ELK.
  In doing this, we
- investigated the two hypotheses.

The framework has been tested in detail on one real example. Most of the techniques are general, applying not only beyond this example, but also outside

the domain of ecology. We are thus optimistic that further testing will prove that the framework works more generally.

The third part of this thesis considers:

1. evaluation of ELK
2. related work
3. future work
4. summary and conclusions

Part III

Discussion

Chapter 8

Related Work / Contributions

8.1 Introduction

The purpose of this chapter is twofold. First, we compare the technical details of the most relevant related work. Second, we identify the precise nature of the contributions of this thesis. The work presented in this thesis is most directly concerned with the problem of representing and constructing ecological simulation models. We have made a substantial contribution to this field. However, as pointed out in chapter 1, we share similar research goals with workers in a variety of in other major fields. In all, the major fields related to this thesis are:

- Ecological Modelling
- AI and Simulation
- Software Engineering
- Intelligent User Interfaces
- Knowledge Representation

In this chapter, we give short reviews of the potentially relevant areas in each of these fields, identifying specific issues and problems that are most directly relevant to this thesis. We identify (to the best of our knowledge) the most important relevant published techniques and/or systems and compare the technical details with the theory and/or implementation of ELK.

8.2 Ecological Modelling

The bulk of published papers in the field of ecological modelling describe specific models of particular ecological systems. However, there is a growing concern about the way that models are constructed and represented. These correspond to two major problems in the state of the ecological modelling art (as noted in chapter 2):

- *Model Comprehension:* Existing formalisms for representing ecological models are inadequate for understanding them. This makes the models difficult to:
  - analyse
  - be used by others
  - modify
- *Model Construction:* Most existing tools for constructing ecological models require considerable mathematical modelling and programming skills. This effectively denies many ecologists the opportunity to construct models.

[Loehle, 1987] gives a strong argument with many examples for the potential uses of artificial intelligence techniques in ecological modelling. Two of his major points directly correspond to the two key problems noted above. Additionally, he notes that:

> "Virtually no research has been published, however, on the application of artificial intelligence techniques to ecological modelling.[Loehle, 1987]"

This paper is purely speculative in nature; it describes what *should* be done, not what *has* been done. Recently, there have been some published reports of how artificial intelligence is being used in ecological modelling [Colomb et al, 1988; Thomson & Taylor, 1990]. However with the exception of the Edinburgh ECO project [Muetzelfeldt et al, 1989], and SIRATAC [Colomb et al, 1988] we have found no work that directly addresses either of the two problems listed above. Therefore we limit the detailed technical comparisons in the field of ecological modelling with these two projects, concentrating largely on the former.

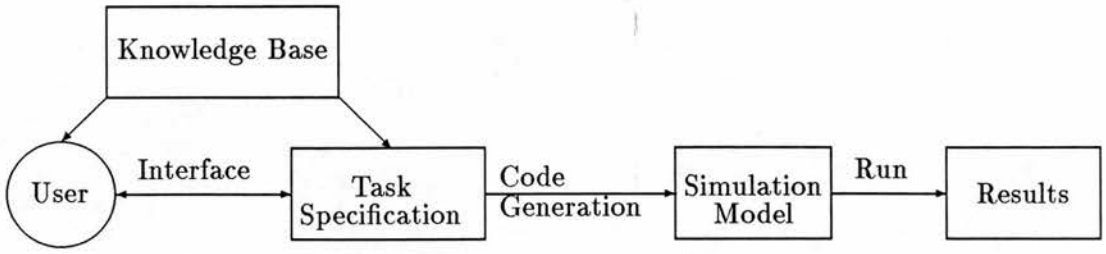


Figure 8–1: The original ECO system

8.2.1 The Edinburgh ECO Project

8.2.1.1 ECO: Introduction

The most recent overview of the ECO project is in [Uschold et al, 1989]. We make no attempt to mention all of the important work related to this project, as much of it is of limited relevance to this thesis. Further details may be found in the cited publications. In the following sections we describe:

- the first generation: ECO
  - original objectives
  - basic functionality
  - shortcomings
- the second generation: EL, SL, NIPPIE
- comparisons with ELK

8.2.1.2 The First Generation

The original objective of the ECO project were to construct an expert system for ecological modelling. The major problems that this system was to alleviate are:

- Many ecologists lack sufficient modelling and programming skills to build ecological models.
- Most models are one-off Fortran programs that are insufficiently documented and thus difficult to analyse and/or assess.
- There is no central store of model components and parameter values that may be accessed and used by others. Thus a large amount of time is wasted repeating the efforts of others.
- There is little standardisation of modelling approaches.

The requirements of the initial system were:

- a *task specification formalism* capable of representing a wide range of ecological models.
- a *front end* which communicates with users in ecological terminology (*i.e.* to reduce conceptual distance)
- a *data base and browsing mechanism* for storing and accessing ecological data and relationships.
- an *automatic checker* to ensure that the models make ecological sense.
- a *back end interpreter* to run the completed model.

A diagram illustrating the general architecture of the system appears in figure 8–1. A task specification formalism, for our purposes, is a language for specifying simulation programs which embody models of ecological systems. Because the range of possible models and programs is far too vast to capture in its entirety, we limited ourselves initially to system dynamics models. This methodology is commonly used in ecology to model the flow of materials such as energy, nutrients, and pollutants. As such, it is quite powerful and capable of capturing a wide range of models. It also has the advantage of being based on relatively few simple concepts.

System dynamics modelling makes use of a concise schematic representation which helps the ecologist think about the model without mathematical formulae. Figure 8–2 shows a diagrammatical representation of a very simple model in which wolves are preying upon sheep. Each box is referred to as a compartment and is conceived of as a tank containing some material. The heavy arrows indicate flow of material between tanks. In this case, the flow might represent the transfer of biomass from the sheep population to the wolf population due to predation. The smaller arrows indicate functional dependency. For example, the rate of predation is a function of the current values for sheep and wolf compartments and a coefficient.

All system dynamics models map directly to sets of differential equations (the converse is not true), one for each compartment in conjunction with non-differential equations for the flow rates. The formalism for representing the models generated using ECO is equivalent to a subset of that described in § 6.3. The direct correspondence of the underlying representation with system dynamics models was instrumental in bridging the gap between the user's view of the problem in ecological terms and the final Fortran simulation program. It enabled the ecologist

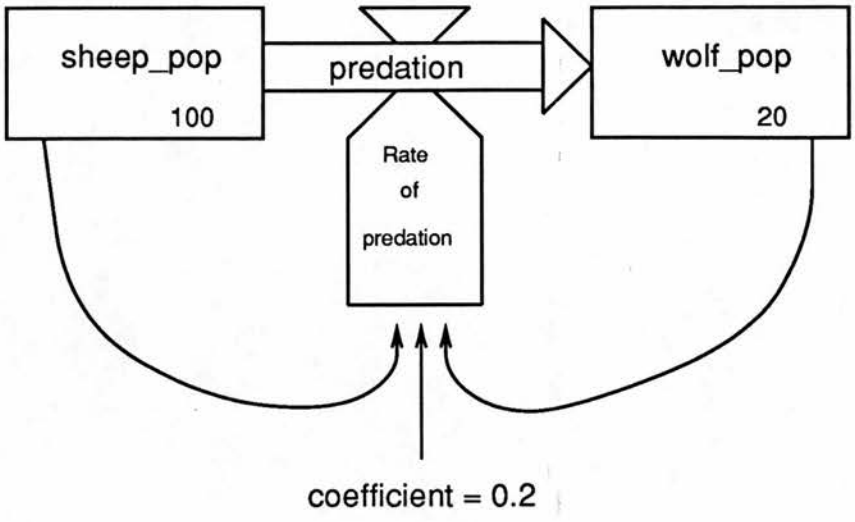


Figure 8–2: A System Dynamics Model

to think in the ecological terms of materials flowing, and expressing their models (via an interface package) into the formalism in a fairly straightforward manner.

Users input ecological statements in a stylised English command language. From the user's point of view, the command "wolves eat sheep" means that there are two populations, wolves and sheep and that the process, eating, causes biomass from the sheep population to flow to the wolf population. The system translates this command into the language of system dynamics resulting in the creation of two compartments and a flow. Other important commands include setting initial values, and selecting equations from a library of commonly used mathematical relationships. The eventual Fortran code is generated automatically from a complete model specification and is never seen by the user.

The first prototype adequately demonstrated the basic concept, but there were many problems. Chief among these were:

- inadequate dialogue facilities
- limited range of models

Much time was spent experimenting with different dialogue strategies. The results of these experiments are reported in [Robertson et al, 1988b].

Our main concern here is the fact that ECO could only construct system dynamics models. Many important ecological entities and processes are not naturally viewed as compartments and flows. Crucially, because

the system relied completely on the system dynamic flow analogy to reduce the conceptual distance

the usefulness of ECO breaks down when the flow analogy breaks down. The conceptual distance issue was finessed by taking advantage of an existing framework whose usefulness is extensive, but not unlimited. *The bridging of the conceptual distance came virtually for free with the system dynamics modelling framework.* That is, there was no real bridge; instead there was a 1-1 correspondence with the user's view of the problem and the underlying representation. There was virtually no distinction between ecological and modelling information. The single exception was the explicit representation of a flow which is used to create the equations, but is not explicitly represented in the runnable specification. In the terminology of this thesis, the flow is part of the ecological description.

The main achievement of ECO was to provide the right amount of syntactic sugar to the representation in the user interface, and to write a code generator which directly translated the formalism into a Fortran program. The functionality of ECO was quite similar to Stella [Lewis, 1986]. The latter is a commercial product with a highly developed graphics interface. The major difference between it and ECO is that the latter had domain specific information. This included:

- ecological objects and processes which correspond to compartments and flows
- a library of ecologically meaningful equations for computing flow rates
- ecological consistency checking

8.2.1.3 Second Generation

The major tasks addressed by the second generation of ecological modelling systems were:

- extend the expressive power of the modelling language beyond system dynamics
- construct a (necessarily non-trivial) bridge to reduce the conceptual distance that exists outwith system dynamics
- enhance the guidance capabilities

That the first requirement gives rises to the latter ones is an important conclusion reported in [Robertson et al, 1988b].

EL

We first describe EL[1][Robertson et al, 1987; Robertson et al, 1988a], the first of the second generation systems on which all the rest are based. To address the conceptual distance problem a new *problem description* language was designed which directly captures the terms that the user is thinking in. Because there was no longer a 1-1 correspondence between what users are thinking about and the model we were forced to relieve the restriction that everything the user says is translated directly and unambiguously as part of the runnable model. Instead, the system was equipped with modelling knowledge and reasoning mechanisms to help the user convert their problem description into the runnable model (*i.e.* the solution description).

EL users are allowed to express facts about the ecological system which may be represented in the runnable model in different ways or not at all. For example, users can say that rate of predation depends on an animal's location. This suggests that a model with explicit spatial representation may be appropriate. Users may choose from different standard ways to represent spatial changes (*e.g.* grid, or zones). This information about the ecological system, as well as information about the simulation model (but not necessarily part of it) is collectively referred to as the *problem description.* The runnable model itself is referred to as the *solution description.* It is a Prolog program, automatically generated from the problem description in conjunction with a library of pre-stored modelling schemata (*e.g.* for the grid method of modelling spatial changes).

EL used sorted logic as a wide-spectrum language for representing all the different kinds of information in the problem description. This language is much simpler than, but shares a common foundation with ElkLogic. Users created the problem description by selecting and editing sorted logic statements translated into English. For example, a generic statement about predation between animals may be selected. This may then be specialised to say that wolves prey on sheep. Formally this results in the following:

$$\forall W{:}wolf.\forall S{:}sheep.\forall T{:}time.predation(W,S,T)$$

which is translated as:

---

[1] EL is an acronym for 'EcoLogic'.

```
For all wolf, sheep and time:
wolf preys upon sheep at time.
```

Including this sentence in the problem description means (in everyday English) that in the ecological system of interest, wolves prey on sheep. Once the user was satisfied that the problem was described adequately, they would select a command to build the model. This required at least one goal of the form: "what is the value of variable X at time T?". The system constructed a program to compute this, thus solving the goal. The program construction technique is based on the Marples algorithm [Bundy et al, 1979]. This works like means-ends analysis and backward chaining. To solve this goal, a schema is selected that can be used to compute X. If new unknowns are introduced, these give rise to new subgoals for which additional schemata are required. This process continues until all subgoals are solved (*i.e.* when there are no new variables, only initial values and parameters). The model consists of an acyclic network of schemata linked by the inputs and outputs. The schema for computing the goal is the root node.

The problem of searching through a large space of schemata is alleviated by associating with each a set of preconditions which must hold before it may be invoked. For example, a schema for using a grid spatial representation method requires that there is a statement in the problem description that says the entities in question vary with location (see [Robertson et al, 1988a] for details of this example).

Another interesting feature of EL is the ability to reconstruct an entire model building session. The basic facility consisted of a mechanism for browsing through the acyclic graph of instantiated schemata. 'At' each schema, a user could see the specific instantiations of the preconditions.

EL constituted a big step in the right direction, however there were serious problems.

1. The program construction algorithm assumes that the problem description is complete, and that there are sufficient schemata in the library. If there are not, then EL stops. Thus there was no way to guarantee that the program would run.
2. Furthermore, once the 'build model' command had been issued, there was no going back to edit the problem description. Such flexibility is essential if the program generator is stuck.

The programs SL and NIPPIE were constructed to address these and related issues and problems.

SL

SL [Robertson et al, 1988a] was constructed primarily to address the first problem. It does so by taking the radical approach of constructing and running the model *simultaneously*. It uses the same program construction algorithm, but instead of starting with a problem description, it builds it up as required on the fly. The starting point is a goal of the same type as for EL. Schema are identified which can compute the goal, and the associating preconditions give rise to questions asked of the user. This is how the problem description is built up dynamically.

Although SL solves the problem of guaranteeing a runnable model, it was highly inflexible. Users had no ability to direct the course of a model building session. Also, the dialogue was difficult to control (this is a classic problem of any backward chaining system).

SL also addressed the blank sheet of paper syndrome faced by EL users who did not know what to to. It took an extreme approach by taking complete control.

NIPPIE

The primary aims of NIPPIE are to:

- enable the program generator to recover from incomplete problem descriptions.
- impose a sensible ordering on the questions asked of the user.

The approach to solving the first problem is to use the basic technique employed by SL of acquiring problem description information on the fly. However, rather than simply ask about each precondition as it is identified as being potentially useful, the system tries to be a bit more clever.

The program generator in SL drives the dialogue in a fairly arbitrary manner. Since it works backward from goals, the preconditions in the initial schema identified will frequently be cast in modelling terms. Thus, one of the first questions that might come up is "is it a spatially represented model?". This goes against the fundamental goal of bridging the conceptual distance. Users should initially be able to talk about ecological things. NIPPIE attempts to identify a series of

ecological questions that could be asked that would gently lead up to the modelling question (*i.e.* the idealisation decision). Great efforts were made to phrase questions in easily understood ecological terminology.

NIPPIE also incorporated some heuristics for ordering the questions in a way deemed to be desirable. The basic principles are (a) ask general questions before specific ones (b) ask related questions together (c) order questions by dependency.

NIPPIE made some significant improvements over EL and SL. In particular, it:

- allows interleaving problem description and program generation phases
- achieves more natural sequence of question asking
- makes an explicit distinction between ecological and modelling information, thus identifying [some of] the idealisation search space.

8.2.1.4 Comparisons with ELK

The major features shared by all four systems are:

- They use a problem description based on sorted logic that:
  - is distinct from the runnable model.
  - facilitates bridging the conceptual gap by allowing users to specify:
    - what is true about the ecological system being modelled.
    - what is true about how they wish to model it.
- There is a facility for representing attribute-based substructure.
- There is a schema mechanism for defining equations to compute variables.
- There is a facility for running the simulation models.
- They can represent models outside the system dynamics framework.

The important features that EL, SL, and/or NIPPIE have which ELK lacks are:

- high-level idealisation rules
- an active mechanism for controlling dialogue where the system may take the initiative

Consider the following rules in EL:

$$\begin{aligned} \exists A{:}attribute.varies\_with(A, location) &\rightarrow spatial\_representation \\ grid\_squares &\rightarrow spatial\_representation \\ zones &\rightarrow spatial\_representation \end{aligned}$$

The first rule says that if some attribute varies with location, then *it is important to consider incorporating explicit spatial representation in the simulation model.* Whether it is incorporated or not is a separate decision. This is analogous to an *_interest* specification in ELK. The term, *occ_interest*(*wb_pred*, *manual*, *heed*), for example, denotes that it is important to consider incorporating explicitly the occurrence of predation denoted by *wb_pred* (*i.e.* between *predators* and *wb_pop*) into the runnable model. The other two rules are used abductively to generate suggestions about how to specify the simulation model. Together these three rules are used to suggest possible idealisation options based on ecological information. Note that the left hand side of the first rule is ecological information, and the right is simulation modelling information. The latter two rules contain only simulation modelling information.

Note that *spatial_representation* is a higher level concept than ELK currently represents. The idealisation in ELK is at a lower level. The need for high-level rules has not arisen yet.

We intend to incorporate these features in ELK. We are optimistic that this will be a straightforward exercise because the underlying representation for all systems is basically the same.

One important difference between the development of ELK versus EL, SL, and NIPPIE is that the logical foundation for ELK is much more thoroughly developed. Important features of ELK that EL, SL, and NIPPIE either do not have at all, or have in a limited capacity are:

- model comprehension
- low level identification and control of idealisation search space
- explicit separation of the problem description into the following three categories:
  - general/ecological knowledge
  - ecological system description
  - dialogue level
- goal elicitation subsystem
- flexibility
  - modifiability
  - user driven dialogue; interleaving tasks
- substructure

- explicit representation for sets
- representation for part-composite substructure

- higher-order functions, especially *average, maximum, etc.*
- user-specified defaults

The first three points are highly interrelated; we discuss them together. EL, SL, and NIPPIE do not distinguish model variables from the ecological attributes and effects of processes that they correspond to. Thus, model comprehension is severely limited. There is no way to record or recover idealisation decisions with respect to model variables. There is no automatic documentation of the ecological meaning of model variables. Because the system does not separate the ecological description from dialogue level information, it is impossible for it to discover implicit idealisations which correspond to something in the ecological description not being in the runnable model. The model reconstruction facility of SL is limited to browsing through the acyclic network of instantiated schemata.

We can tease out the distinctions in terms of our knowledge ontology. For example, the sorts and uninstantiated sentence templates are general/ecological knowledge. Entities (called 'objects' in EL) and instantiations of the templates are part of the ecological description. A predication like *spatially_represented* constrains how the model is to be built by being a precondition to various schemata, but is not part of the runnable model. Thus it is at the dialogue level. Because the system is not explicitly aware of these distinctions it can not make use of them in the way that ELK does. As discussed above, this severely limits the facilities for identification of idealisation search space and model comprehension.

A fundamental design constraint for ELK is flexibility. The wholly system-driven approaches take by NIPPIE and SL are inadequate. EL and ELK both are wholly user-driven in the problem description phase. EL however, is unacceptably inflexible when it comes to constructing the model. Users can not go back and change their mind. NIPPIE solved this problem, but took away the control from the user.

ELK has been designed to achieve the best of both worlds with respect to flexibility. The key idea is to provide a suggestion box mechanism similar to that used in EL. Users could then do what they like, but would also be free to take advice from the system as required. Users who always followed this advice would

achieve the effect of a wholly system driven approach. However, they are not forced to do what the system says like in SL and NIPPIE.

As with everything else in ELK, the user is in complete control when selecting schemata. This is a significant departure from both EL and NIPPIE. However, unlike EL and NIPPIE, ELK currently cannot drive the schema instantiation process. So with respect to schemata, the job is half done. The last step will be to incorporate dialogue control mechanisms like those in NIPPIE. The exact same heuristics used by NIPPIE for deciding what to do next may be used in ELK. The difference will be that the control information will be presented as a choice, not an edict.

One important feature missing from all of the other systems is the rich representation for sets and the induced functions defined on sets. EL has a special sort called *population*. All populations of all sorts of animals have this sort. In ELK, the set type *set(animal)* is used instead of *population*. There is of course nothing to stop an ELK user from explicitly defining a sort called *population*, and creating the entities *ln_pop* and *predators* as instances. Users who understand the fundamentals of the representation used by ELK will know that by doing so they will be unable to make use of the many facilities ELK provides with respect to the distinction between sets and individuals. All the attributes would have to be manually specified like *biomass*, *age*, etc. as well as induced ones using maximum, average, etc. In ELK, this is done automatically.

8.2.2 Siratac

SIRATAC [Colomb et al, 1988] is a large successful commercial simulation package which advises cotton growers whether, when, and how they should spray their crops. It uses conventional rule-based techniques in conjunction with various underlying simulation models to provide advice. The rules analyse the output of the simulations.

They are not concerned with assisting in model construction. However, they are very much concerned with making the model more usable, understandable, and modifiable. The initial program was 30,000 lines of Fortran. It was not built using good software engineering practices. It suffered badly from being hard to modify and understand. It has since been reimplemented, and was scheduled to go on-line in the 88/89 growing season.

The techniques used to ensure modifiability differ significantly from ELK in kind and specific purpose. They use a so-called knowledge dictionary based on an entity-relationship model to store information about the model. However, they concentrate on representing in a queryable data base the information in the rules, rather than in the simulation models. No attempt to record the meaning, assumptions, and idealisations of the simulation models appears to have been made.

8.3 Artificial Intelligence and Simulation

We are concerned with quantitative (not qualitative) and continuous (not discrete) simulation. There is some work concerned with mixing the continuous and discrete paradigm which is relevant only to future work on ELK. The nascent field of AI and simulation may be divided into two main categories [Widman et al, 1989]: how can artificial intelligence techniques be used to augment simulation technology and vice versa. We are only concerned with the former; the main AI 'technique' we use is logic-based knowledge representation and inference. There are many aspects to the simulation life-cycle where artificial intelligence techniques may be applied. These include:

- model comprehension
  - assumptions on which the model is based (*i.e.* idealisations)
  - the meaning of model components
  - dynamic behaviour of simulation
- construction and subsequent modification of models
- verification and validation
- planning experiments to achieve simulation goals
- analysis of simulation results, possibly suggesting model revision

There is very little work on:

- using goals
- providing intelligent assistants for the initial construction of models
- enriching representations to facilitate model comprehension.

In the subsequent sections, we discuss three systems which have begun to address some of these issues, and compare them to ELK.

8.3.1 Knowledge-Based Simulation

Since 1980, KBS [Fox, 1986] has been a testbed for exploring the application of AI to the simulation life cycle. There is significant overlap in their goals and those of the ECO project.

> "Industry has been slow in adopting simulation as a means for analyzing complex decision problems. One reason is that the complexity of the modeling language and the differences between simulation modeling concepts and the system to be modeled [*i.e.* large conceptual distance] make model building a difficult and time-consuming task. Early work in *knowledge-based simulation* has attempted to ... create simulation models which are
>
> - explicit,
> - understandable,
> - modifiable, and
> - self-explanatory.
>
> By using a frame language to represent domain concepts, such as object structure, and goals, there is a one-to-one correspondence between the domain and the simulation model. ... [Fox, 1986]"

They have restricted their work to discrete simulation domains such as factory scheduling and project management. This choice of domain is particularly judicious with respect to model construction, representation, and comprehension. In this domain, frame languages are particularly well suited, rendering the conceptual distance relatively small. This is directly analogous to the first ECO prototype which also enjoyed a one-to-one correspondence between the system being modelled and the modelling language. With such correspondence, the conceptual distance issue goes away. Difficulties with model construction and comprehension are dramatically less. Thus the most important and advanced aspects of their research *do not* center around the key foci for ELK. Whether or not there were more challenging real systems to model for which the one-to-one correspondence no longer held, no attempt was made to solve those problems. Rather, the emphasis of the KBS project was to address *all* the major aspects of the simulation life cycle mentioned above.

The primary purpose for building models in KBS is either to verify a hypothesis, or optimise one or more features of a system. Goals play a key role here. For example, a goal might be to maximise utilisation of a machine. A goal language

was developed which due to the nature of the problem is quite different from goals in ELK. Measuring 'instruments' are set in place to gather data. The simulation is run, the data analysed, and performance summaries are given. A rule-based component is used to reason about how the model might be changed to improve the goals.

Their use of goals is quite different from ours. We use them to provide hooks into the model construction process. They use them to identify simulation experiments to be performed. We both use them to determine to some extent the nature of the output of the model.

Summarising, KBS is a state of the art application of artificial intelligence to the simulation life cycle and is recognised as being one of the first and most extensive efforts [Widman et al, 1989]. However, the main foci of that project are different from ours.

In the ecological domain the problems of representation, construction, and model comprehension appear to be much more difficult than in the factory domain. For example, there are many different levels of organisation in ecology, and the overall idealisation problem is much more severe. A key observation is that unlike the factory domain there is no convenient 1-1 correspondence between the model and the way an ecologist thinks about the ecological domain (assuming we do not limit ourselves to system dynamics models).

This means that having a small conceptual distance is no longer for free, we must work to achieve it. It also means that model comprehension does not come for free. The need to record explicit idealisation decisions it not addressed by the KBS project, possibly because the need does not arise in the factory domain. This was a major focus of the ELK project.

8.3.2 Object-Oriented Language for Continuous Simulation

[Lounamaa, 1986] describes a language (SLICL) for continuous simulation models which is very similar to that described in § 6.3. I quote:

> "Let $x(t)$, $y(t)$, and p denote state variable, dependent variable, and parameter vectors respectively. Then the difference equation models are of the form
>
> $$x(t+1) = f(x(t), y(t), p), \qquad (8.1)$$

$$y(t) \;=\; g(x(t), y(t), p). \qquad (8.2)$$

The equations defined by (8.2) are restricted to be nonsimultaneous such that they can be solved in one pass without iteration. That is, the dependencies between the dependent variables have to be acyclic, in particular, the right side of the equation that determines the value of a specific y variable cannot contain the variable itself."

What are referred to as dependent variables here are either intermediate, partial rate, or exogenous variables in ELK. SLICL is an object oriented representation defined by various Lisp macros. State variables, dependent variables, and parameters are defined using constructs extremely similar to the *_variable* constructs in ELK.

The primary goals of this project are shared with ELK.

- to reduce conceptual distance
- modifiability

The author [Lounamaa] views this as a high-level modelling language which runs as fast as an equivalent Fortran program. Objects correspond to entities in ELK, each is assigned any number of state variables, dependent variables, and/or parameters. There is no separate notion of partial rates for incrementing and decrementing state variables each time step. We agree that this is a significant improvement over Fortran. The author notes that a graphic interface is required, but not currently implemented. If that were added, the result would be a tool like Stella with increased expressive capability because it would not be limited to system dynamics models.

The representation described in [Lounamaa, 1986] is a notational variant of ELK's runnable-model level constructs. If the claims about efficiency are accurate, this might be a viable alternative to Fortran as a target language. However, the SLICL research has addressed only the problem of developing a higher level language for specifying runnable models. This issue was not of interest in the design of ELK; ELK merely used the solution from the original ECO program [Uschold et al, 1986].

8.3.3 Planetary Atmospheric Modelling

[Keller et al, 1990] describes a project very similar to ELK in the domain of planetary atmospheric modelling.

> "In general, implemented scientific models are complex, idiosyncratic, and difficult for anyone but the original scientist/programmer to understand. We believe that advanced software techniques can facilitate both the model-building and model-sharing process."

They note the following as important barriers to scientific model-sharing (*i.e.* reuse).

- "*Lack of comprehensibility*": They assert that models usually are complex, difficult to understand, and insufficiently documented. Even well documented models inevitably cease to be so as the model evolves over time.
- "*Wrong level of abstraction*": This is their term for 'large conceptual distance'. "...the model builder is forced to translate a model originally expressed on paper in terms of natural scientific concepts into a 'foreign language' ...". The foreign language they refer to is Fortran.
- "*Unmodifiability*: Due to problems with comprehensibility and level of abstraction, models are often difficult to change to suit the needs of a different scientist."
- "*Implicit assumptions*: Often important modeling assumptions and data assumptions are left implicit in the low-level code that implements the model." These assumptions may neither be inspected nor changed by a new user. "This is a significant deterrent to using another scientist's model because the appropriateness of assumptions is frequently the source of scientific debate."

The major goals are identical to those of ELK. The techniques that they propose are:

- "*Interactive graphical interface:* to enhance comprehensibility and modifiability of models. ..."
- "*High-level modeling language:*" to reduce conceptual distance
- "*Analysis facilities:*" to interpret model output
- "*Intelligent Assistance:* ...constraint satisfaction, typed inheritance hierarchies, and backward chaining control can reduce the amount of detail that a scientist-user needs to track."

Overall, the similarities between this project and ELK are striking, both in the analysis of basic problem being addressed and in the techniques for solving them. An initial prototype has been constructed which may be used to create what in

ELK is the computation network. "For the purpose of our initial prototype, we have conceptualized model-building as a process of linking uncomputed physical variables ... to computed variables". It works in exactly the same backward chaining fashion as do EL and NIPPIE. Menus of variables and schemata to choose from are presented much as in ELK.

The major difference between this project and ELK, is that [Keller et al, 1990] describes a proposal, not a project that is substantially underway.

8.4 Software Engineering

In ELK, we are concerned ultimately with creating runnable specifications of ecological simulation models. This constitutes a large class of software application programs. Thus, our work is directly relevant to software engineering, the following areas in particular:

- domain modelling
- software comprehension
- reuse
- requirements capture

Research on providing computer assistants in the domain of requirements capture is relatively new, owing largely to the fact that creating requirements documents for software systems is a highly complex task.

The latter three areas are tightly interrelated. One of the key barriers to software reuse is that most complex software is difficult to comprehend. Use of domain models enhances software documentation. This improves software comprehension and thus facilitates reuse.

8.4.1 Domain Modelling, Software Comprehension, and Reusability

Software comprehension is the general term for what we call model comprehension in ELK. This is also referred to as program understanding [Biggerstaff, 1989], and *discovery* [Devanbu et al, 1990]. The concept is a simple one, although difficult to achieve satisfactorily. In order for software to be understood properly, an explicit

record must be made of important design decisions, and each component should be explicitly related to some aspect or feature in the domain of application.

LaSSIE [Devanbu et al, 1990] is a software information system whose primary aim is to reduce or eliminate invisibility. This is a serious problem for large software systems; programmers spend huge amounts of time discovering what the code is doing before they can modify it. This drastically reduces the possibility of software reuse. This is precisely the same problem with many simulation models. The approach to reducing invisibility is to build a model of the domain which can be queried by programmers to locate quickly the information they require about the software. LaSSIE incorporates multiple views of the software, including

- *an architectural view*
  e.g. This part of the system is part of that layer of the system architecture.
- *a conceptual view*
  e.g. This part of the system affects those conceptual objects in the domain.
- *a code view*
  e.g. How do the source files and declarations relate to each other?

Making such distinctions and equipping the knowledge base accordingly greatly facilitates understanding of the software. Similarly, in ELK, the distinctions we make in our ontology facilitate understanding of the simulation model. Take the wildebeest population as an example. The runnable-model (*i.e.* code) view is the variable *n_wb*. The ecological view is the attribute *number* of the entity *wb_pop* which is a set of entities of sort *wb*.

Some important differences between ELK and LaSSIE are:

- LaSSIE is concerned with describing existing large scale software about which there is generally insufficient knowledge. This knowledge must be acquired through a process of reverse engineering from the existing system. ELK on the other hand, builds models from scratch. We do however, share the ultimate goals of software comprehension which in turn facilitates modifiability and reuse.
- LaSSIE is intended for use by programmers, ELK is intended for use by non-programmers.

8.4.2 Ignoring Attributes

An important fact about ELK is that we represent many more attributes in the general/ecological knowledge base than will ever be used in a particular model. The idea of representing more about a domain than will be used in a particular application is also relevant in the use of domain models in software engineering. For example:

> "Current object-based systems view specialisation exclusively as a way of saying more about a class. For a corporate domain model it will also be necessary to specialise a class by saying what properties of the superclass are not relevant to the subclass. For example, shop floor worker would logically be a subclass of employee, which could be used by a factory control system. However, while address would be a logical property for an employee, it might be *of no interest* [my italics] to the factory control system. Thus the specialisation from employee to shop floor worker would include the information that the address attribute should not be included on a shop floor worker [DeBellis, 1989]."

[DeBellis, 1989] is purely speculative; no solutions are offered. The ignoring of attributes described there is rather stronger than the sense in which we ignore attributes in ELK. For a start, the 'ignoring' referred to above is a fixed part of a domain model; ELK is interactive. It does not remove attributes entirely, but rather gives the user the option as to whether and how they wish to use certain attributes. The above scenario would be more directly analogous if it was known *a priori* that an inherited attribute of an entity was never appropriate. We have not catered for this, although we recognise its potential usefulness. For example, in population dynamics models, the attribute 'length of hair' would almost never be of interest.

Instead of ruling attributes out entirely, the ELK approach is to order them so that those unlikely to be used come last when an attribute must be selected from a menu. General information about the type of model (*i.e.* high-level goals as discussed in chapter 3) could be used to prioritise these menus. For example, a future version of ELK may allow users to input a high-level goal like: " to have a population dynamics model of ...". This would give low priority to the attribute 'length of hair' but high priority to *number*. This is another potential use of goals.

8.4.3 Requirements Capture

8.4.3.1 The Requirements Apprentice

The role of the requirements apprentice (RA) [Reubenstein & R., 1989] is to assist the requirements analyst in specifying a formal requirements document. The problem is particularly difficult because initially, the analyst does not know exactly what they want. The process begins with informal, ambiguous often conflicting requirements. The purpose of the RA is to allow such statements to be made at the outset, and to gradually remove the ambiguity and increase the formality. The RA produces three kinds of output:

- Interactive output gives the user feedback on the consequences of their actions.

  This is analogous to the messages produced by ELK.
- A *requirements knowledge base* represents everything the RA knows about the current state of the requirements.

  This is analogous to the collective base of specifications at the ecological system, dialogue and the runnable-model levels.
- Various written documents can be generated from the requirements knowledge base (not implemented yet).

  This is analogous to the automatic documentation produced by ELK.

Other important similarities with ELK include:

- recognition that significant amounts of domain information is required in order to provide an adequate level of assistance
- the analyst [*i.e.* user] is in complete control
- the knowledge base is never complete, and will require dynamic updating
- significant effort is made to convert informal information into a formal specification, bridging a considerable conceptual gap.

The techniques employed to go from informal to formal differ. In this regard, the RA is more flexible. The RA lets the user say things which are not defined yet, which may have different interpretations. As additional information is specified, the RA attempts to resolve the ambiguity. Currently, ELK provides no such facility; users are always forced to define their terms. The assistance that ELK provides in this regard is to provide a sequence of constructs which users may choose from which bridge the informal–formal gap. The knowledge ontology plays the key role.

The RA relies more heavily on inference. Currently, the RA can take several minutes to respond to certain commands when it tries to do inference. Response time in ELK is never more than a few seconds. Insufficient details are provided to ascertain why the inference demands for RA appear to be so much greater than for ELK. One possibility is the dynamic ambiguity resolution may be very costly. Currently, the user interface for the RA is not implemented.

8.4.3.2 KATE

KATE [Fickas, 1987] is another system for assisting in the requirement capture process. Although the overall goals are very similar to both the RA and ELK, different aspects of the problem are being tackled. Fickas is concentrating on the design aspect, and is developing techniques for critiquing the design, and for resolving conflicting design goals. We are not concerned with this, so there is little to directly compare with ELK.

8.5 Intelligent User Interfaces

There is no coherent body of research that constitutes a subfield concerned with making user interfaces more intelligent. We have found no comprehensive general overviews of the 'field', which clearly describe the current issues and problems and the state of the art. [Rissland, 1984; Bundy, 1984] are steps in the right direction, but are neither comprehensive nor recent. There appears to be no particularly seminal papers, and no coherent group of workers who reference each other.

Instead, there are a great many papers describing various systems which purport to include some form of intelligence in some kind of interface [O'Keefe, 1985, Uschold et al, 1986, Barstow et al, 1982, Bennet & Englemore, 1979, Martin et al, 1983, Ross et al, 1985, S. et al, 1982] Some generally important issues that we are have not addressed in ELK are:

- user modelling (a field unto itself)
- natural language interfaces to data and knowledge bases.
- adaptable interfaces which vary according to the performance of users.
- degree of coupling of front end and back end

The important issues that we have addressed that are of general applicability are the use of knowledge based approaches for:

- reducing conceptual distance
- identification and pruning of choices
- consistency checking
- relief from redundant and/or menial tasks

A highly relevant work which influenced this research is [O'Keefe, 1985]. The goal of that project is to build an intelligent assistant for statistical analysis, called ASA. The key problems addressed in that project that are similar to ELK are:

- representation of a complex domain with many subtle distinctions
- ability to communicate with non-expert users in their own terms

The representation for value spaces in ELK is a simplification of that used in ASA. In ELK, a value space is just a set of values. This set may be represented by a sort (*e.g.* *integer*), or an instance of a set type (*e.g.* $\{1,2,3\}$:*set*(*integer*)). In ASA, they are much richer. In the statistical domain, they are much more important than in the ecological modelling domain. Of critical importance is knowing what operations make sense. For example, although temperature is measured in numbers, it makes no sense to add them, however differences are quite relevant. Specifically, in ASA:

- value spaces are arranged in a lattice; properties are inherited from parent nodes.
  e.g. addition on integers in inherited from reals
- physical dimensions such as length, mass, etc are incorporated
- operations may be performed on the value spaces themselves
  e.g. length can be used to derive volume.
- users may initially be vague about what a value space is and gradually specify more information.

There were many problems that were addressed in ELK that were not addressed in ASA. For example,

> "... there is an enormous man/machine interface problem that fell completely beyond the scope of this research [O'Keefe, 1985]"

In ELK, the representation language was carefully designed to make the interface problems manageable. A major emphasis in the ASA project was on computational efficiency; this was not an important issue for ELK.

The representation for sets and substructure was emphasised in ELK, but not ASA. Although parts are cited as being important in statistical domains, the representation used was fairly conventional (see § 8.6.2). Sets of objects, and measures of average, mode, maximum etc are of primary importance in statistics, however there was no attempt to represent these as was done in ELK.

8.6 Knowledge Representation

8.6.1 Structured Object Languages and Tools

The fundamentals of ElkLogic have much in common with many structured object based languages and tools (*e.g.* [Clayton, 1984; Bobrow & Stefik, 1981; Kee1986]). See [Stefik & Bobrow, 1985] for a good overview of basic issues in object-oriented programming. The similarities include:

- A specialisation hierarchy (frequently called an *isa* hierarchy of classes)
- A distinction between:
  - sets of things of a certain kind (usually called *classes*)
  - individual things of a certain kind (usually called *instances*)
- Slots (also called attributes, roles, or variables) associated with
  - classes (*e.g.* maximum height)
  - instances (*e.g.* height)
- Inheritance of slots
  - from classes to subclasses
  - from classes to instances

There are many features of object-based tools that are not included in ELK. These include:

- multiple inheritance
- methods and messages

Given the close mapping between ecological systems and object-oriented representations, it is natural to consider using object-oriented programming to model

ecological systems directly (*e.g.* as factory systems are modelled in KBS [Fox, 1986]). Others are doing exactly this [Saarenmaa et al, 1988; Folse et al, 1989].

To avoid the criticism of reinventing the wheel we must justify why we did not take this approach. We shall compare ELK with KEE. Although other tools provide different features and capabilities, the major points we have to make are valid for all object-based tools that we are aware of.

In KEE, everything is a *unit*. Two fundamental kinds of units are classes and instances. These correspond to sorts and entities respectively in ELK. What we call attributes in ELK are referred to as slots, their properties (*e.g.* value spaces, default value) are referred to as *facets*. Instance units are associated with class units via the *instance* relation. Instances inherit the slots from their class, as well as those of superclasses. This is exactly as in ELK. We shall discuss two essential requirements of ELK which are not supported adequately by current object-based languages and tools. These arose in the context of ecological modelling, but are not specific to this domain.

- *rich representation for sets*
- *ignoring attributes*: to selectively ignore attributes which are known to exist but are not of interest

8.6.1.1 Sets

The design requirements for ELK with respect to sets are summarised below:

1. distinction between classes and instances
   e.g. shp is an instance of the class *sheep*
2. instances (*i.e.* members of classes)
   - may be freely added and removed
   - are not required
3. the attributes that apply to a certain kind of thing are distinct from the attributes that apply to sets of things of that kind
   e.g. height versus *maximum_height*
4. many of the attributes that apply to sets of a certain kind of thing are derived from the attributes that apply to that kind of thing.
   e.g. maximum_height derived from *height*
5. representation distinguishes between:
   - the set of all things of a certain kind (*e.g. sheep*)

- a set of things all of which are the same kind (*e.g.* flk:*set*(*sheep*))

Requirements 1 and 2 are standard with almost any object-based language. In KEE, things of a certain kind are usually represented as instances. Sets of things of a certain kind are usually represented as classes. Many tools do not support requirement 3 explicitly, (*e.g.* Art, Knowledge Craft) although KEE does. KEE distinguishes between class variables and instance variables[2]. The former apply to the class (*e.g.* maximum height), the latter only to instances (*e.g.* height). The main use of this distinction is to stop inheritance of class variables to instances. This is exactly what we need in ELK. Art and Knowledge Craft do not make this distinction explicit; instead a more powerful facility for specifying which slots are inherited and which are not is provided. In any event, users must manually specify every class variable either explicitly as such, or by suitably specifying the inheritance behaviour. In ELK, a whole range of class variables and the appropriate inheritance behaviour are created and specified automatically. Members are freely added and removed by modifying the instance relation. There is no requirement that there are any members of any particular class.

We consider how two flocks of sheep $flk1$ and $flk2$ might be represented (see § 5.3). We identify two 'obvious' ways that we might do this in KEE and discuss their shortcomings; then we describe the ELK solution. The first is to represent the flocks as classes. KEE automatically keeps track of what the member instances are. Member sheep (if any) may be freely added and/or by creating/destroying instances of these classes. To avoid duplication of specification of attributes, it would make sense to create a superclass of $flk1$ and $flk2$ called *sheep*. So, *number* might be a class variable for *sheep*, and *height* an instance variable; both are inherited by $flk1$ and $flk2$.

This solution fails to meet requirement 4 because there is no facility for inferring the class variable 'average height' from the instance variable, 'height'. It is likely that this could be done, but specialised Lisp code would be required. However, because average, maximum, etc can be nested arbitrarily, there are infinitely

[2] In KEE, instances are called *members*, class and instance variables are called *own slots* and *member slots* respectively. I sometimes avoid using the specific KEE terms because doing so adds unnecessarily to the 'term soup'. The terms 'class variable' and 'instance variable' are used in [Stefik & Bobrow, 1985].

many variables that may be inferred this way. This would have to be controlled. Also, even if we were to limit the automatic slot creation to a single level (neglecting 'maximum average height' etc) there would still be the problem of lots of unnecessary slots being created since at any one time, relatively few of the induced attributes would be used. ELK circumvents this problem by implicit specification.

It fails to meet 5 because the flock classes are unlike most classes in that the members in both classes are all the same kind. Thus, we are forced to overload the use of class. This means that users will have to be careful to keep track of the different ways that they are using classes. Also, it is counterintuitive, as discussed in § 5.3.

To avoid the overloading of the use of classes, another alternative is to represent the flocks as instances ($flk1$ and $flk2$) of the class *flock*, which might be a subclass of *population* (this is what EL does [Robertson et al, 1987]). Each flock instance would have a slot called *members* (inherited from *population*). For example, the value of the members slot of $flk1$ might be $\{s_1, s_2, s_3\}$. This slot has a value space which is a list. KEE provides a facility for restricting the items in the list to be instances of the class *sheep*. This solution avoids the problem of overloading the use of classes, but is a worse solution that the previous one in terms of the other requirements. The previous solution exploits the system's instance relation to create members; this is used to record the fact that the members of the flocks are sheep. In the alternative, there is no explicit connection between *flock* and *sheep*. The fact that the value of the *members* slot is constrained to be instances of class sheep, is purely incidental. The same instance could have another slot called *foo* whose value was constrained to be a list of instances of fish and chip suppers. As far as KEE is concerned, there is no difference between the attributes *members* and *foo*. This means that the user will have to do more work to keep track of what is going on. For example, the value space for *members* would change for each subclass of *population* (*e.g.* a *pride* would have lion members). As with the first solution, this one offers no support for deriving the attributes that apply to flocks from the attributes that apply to sheep. There would be two different mechanisms for listing members of sets: the usual one for instances of classes, and a separate one for listing values of slots which are being used in this special way. This can be regarded as a feature which helps keep distinct the two ways that sets are being used.

Because of the importance of sets and the different ways that they are used in the ecological modelling domain, ELK is purpose built to meet the above requirements. It suffers from none of the above problems. The second solution is closer in spirit to the ELK solution to the problem of representing the flocks. In ELK, we create the sort *sheep* and assign it attributes. This automatically induces the sort $set(sheep)$ which is exactly what *flock* is supposed to be representing. The two flocks are instances of $set(sheep)$ rather than *flock*. Furthermore, if $sheep \sqsubset^s animal$, then $set(animal)$ is also induced. This corresponds precisely to what *population* is supposed to represent. What is being represented by the slot *members* is accomplished in ELK using the component relation. Thus, $s_1 \subset_o flk1$, $s_2 \subset_o flk1$, etc.. The information carried in the restriction of the value space for *members* is embodied in the fact that for all sorts S, $S \subset^p \#S$. There is no need to worry about different value spaces for members of lions or sheep because this attribute does not exist in the ELK solution. The main benefits of ELK derive from the fact that the most set related concepts are automatically induced. This includes set types (*e.g.* $set(sheep)$) as well as attributes of these types (*e.g.* $qnam(average, height)$). It is important to note that:

The type function 'set' is the basis for meeting requirements 3-5.

This enables distinguishing between the following:

- sets which consist of all things of a certain kind
  (denoted by sorts, *e.g. sheep*)
- sets all of whose elements are of a certain kind
  (denoted by instances of set types, *e.g.* $flk1{:}set(sheep)$)

set also plays a central role in the inducing of attributes of sets from the attributes of the kind of members. This is manifest in the typing of *average*, *maximum*, etc.

8.6.1.2 Ignoring Attributes

Even if KEE were to be equipped with the appropriate special machinery for handling sets and inducing their attributes automatically, there is another requirement that arises in the context of modelling that is not catered for; the ability to ignore attributes. For any given entity in the ecological system to be modelled:

- there will be on the order of up to a few dozen basic attributes that apply
- there will be infinitely many induced attributes

- relatively few of its attributes will be used

However, all of the attributes must be represented in the general/ecological knowledge base so that:

- they may be browsed through by users and/or presented in menus for
  - creating ecological variables *or*
  - defining substructure
- idealisation decisions of the form "we are ignoring something which is potentially important for this particular simulation model" may be recorded to ensure model comprehension.

In short, we need to represent lots of attributes but have the facility to ignore them as required. The main reasons are to facilitate identification and recording of idealisation decisions. Their existence gives rise to the identification and pruning of the options for deciding which attributes are used to define substructure or ecological variables. The record of the selected attributes is explicit, (in the *_var* specifications); the record of the ignored ones may be implicit or explicit depending on whether the user created an *att_ent_interest* specification and set the *ig_spec* slot to *ignore*. In all current object-based tools that we are aware of no such facility is catered for; a class either has a slot, or it does not.

A crucial aspect of ELK *is the explicit separation of the description of the ecological system and the simulation model.* Using an object-oriented language in the obvious way for modelling ecological systems directly would necessarily fail to record the idealisation decisions that we need. To accomplish this requires a separate level of information which in ELK is the general/ecological knowledge base. To achieve the separate level would require having a separate knowledge base which was used to guide the construction of the simulation model. Thus, we would have to reimplement ELK in the language. We can only speculate on the ease with which this could be done. At best, it is likely to give rise to much redundancy which ELK avoids because of extensive use of implicit specification. At worst, it would not be possible.

8.6.2 Substructure

Recall that by *substructure*, we mean to include the set-theoretic relations of membership and subset as well as the part–composite relation. Strictly speaking, this

includes the instance and subsort relations. However for the purpose of this thesis, we exclude this from our use of the term 'substructure'. This distinction is formalised in ElkLogic. Although both $shp : sheep$ and $shp \subset flk1$ denote the member-set relation, and both $sheep \sqsubset animal$ and $\{s_1, s_2\} \subset flk1$ denote the subset relation when we refer to substructure, we mean to exclude ':' and '$\sqsubset$'.

Like intelligent user interfaces, there appears to be no coherent body of research that constitutes a subfield concerned with substructure. We have found no general overviews of the field, and no single coherent group of workers who share common goals and reference each other. In perusing literature, there seemed to be two distinct kinds of work being done each with rather different goals.

1. Building practical systems [Kim et al, 1987, Blake & Cook, 1987, Rothenburg, 1989, Kuczora & Cosby, 1989]
2. Building formal theories [Bunt, 1986; Hayes, 1986; Link, 1983]
   - representing the mass-count distinction in natural language
   - common sense reasoning

We discuss these two kinds of work in turn.

8.6.2.1 Practical Systems

There is a growing awareness that better support is required for representing substructure. For example:

> Most object-oriented systems support only minor variations of the "class-subclass" (also called "IS-A" or "taxonomy") relation along with a corresponding "inheritance" mechanism to maintain taxonomic relationships (*i.e.* specialized inferential support for the class–subclass relation). We are attempting to provide a true multiple-relation environment in which different kinds of relations are supported by appropriate specialized inference mechanisms and to provide a general facility to allow the simulation developer to define new relations with appropriate inferential support. [Rothenburg, 1989]

They give various examples of special relations requiring support including *part_of*. The same point is made in [Jang, 1988].

With a few exceptions, [Blake & Cook, 1987; Bobrow & Stefik, 1981] there is no support at all for the part-composite relation. This is a quite surprising state of affairs given the widespread applicability, especially in engineering domains. A standard way to represent parts in an object-based language is to create a slot

called *part_of* for an object. The value space may be a single object (*e.g.* head of an animal) or a list (*e.g.* wheels on a car). If *part_of* is a slot for many objects, then it is *de facto* a relation. Some tools support relations explicitly, [Clayton, 1984; Kno1988] allowing properties such as transitivity to be defined. Some will automatically define the inverses (*e.g. has_parts*) which is useful; many do not. Some tools will allow these relations to be displayed using the same mechanism as the system defined class–subclass relation. Failing this, users will have to write their own display routines.

Some tools provide explicit support for *part_of*. In Loops [Bobrow & Stefik, 1981] for example, you can send a *new* message to some predefined composite object class and it will recursively create an instance of the whole as well as instances of all its parts and make the required *part_of* connections. However the following facilities that ELK provides are not available on any tool that we are aware of:

1. a uniform component relation which incorporates member of a set and subset in the same hierarchy as part-composite ($\subset$ in ELK)
2. a mechanism for type checking so that a user will be prevented from specifying substructure that does not make sense. ($\subset^p$ constrains what may be specified in $\subset$)
3. a capability for ignoring parts that are not of interest

We have discussed the first two points at length in previous chapters. The latter requirement is analogous to the need to ignore attributes. In all tools and languages that we are aware of, a tree is a tree is a tree and every tree has exactly the same attributes and parts (if parts are supported)[3]. They can of course have different values for the attributes (instance variables). For ecological modelling, potentially each tree or group of trees may be represented differently. It is important to be able to experiment freely with different representations. To do this, you do not want to have constantly to change the definition of a tree. Everything ELK 'knows' about a sort (*e.g. tree*) is determined by the attributes and parts it has in the general/ecological knowledge base. For example:

---

[3] By *tree*, we mean a plant, not a data structure.

$$att\_def(biomass, tree, \ldots) \quad att\_def(height, tree, \ldots)$$
$$trunk \prec^p tree \qquad branch \prec^p tree$$

In ELK "a tree is a tree is a tree and it always has attributes biomass, height etc. and the parts trunk, branches etc.." *only in the general/ecological knowledge base.* Because of ELK's multi-stage modelling, not every tree in the description of the ecological system need have branches, trunk, explicitly represented, although it will always have the same attributes. The attributes and parts in the ecological system description may or may not be explicitly represented in the simulation model.

Of course, this flexibility when not needed could be a nuisance forcing users manually to specify the same parts for several entities of the same kind. If a user always wanted certain parts (and certain numbers of them) they should be able to specify this. We have partially implemented such a facility for attributes (*e.g.* allowing automatic creation of model variables). An analogous facility for allowing automatic creation of parts could also be provided. Users would then be able to specify that every sheep had one head and four legs, but no tail. Other users might only be interested in the tails.

The ability to ignore parts and attributes is essential in the ecological modelling domain because it will rarely if ever be the case that all attributes and parts of all entities in the ecological system are used in a particular simulation model. Even if every entity of a certain sort had the same parts for a particular model (*e.g.* all sheep have four legs), it is unlikely that all parts would be necessary.

In some domains these facilities may never be useful. For instance, if you are creating a window graphics package it is extremely important for all objects of a certain class to be the same.

In domains where there are many complex composite objects but rarely sets of objects, then the uniform framework for sets and substructure will be of little use. Existing mechanisms for handling composite objects would be adequate. The domain of ecology is rampant with sets and substructure. Below we describe two systems which have attempted to provide some support for the part–composite relation.

[Blake & Cook, 1987] describes an extension to Smalltalk [Goldberg & Robson, 1983] which allows part hierarchies. One encouraging result was that parts are compatible with the existing multiple inheritance mechanisms. There were

problems too. They do not recognise the similarity between the part and element relations; rather they *explicitly deny it.* For example, a section heading in that paper reads: "Elements of a Collection are not Parts". This is the same point made in [Bunt, 1986] (discussed below). They use the term, structured whole, or just whole, but they mean the more restricted sense of a whole which we call a composite. They go on to say that their formalism had significant problems when there were collections of wholes (*e.g.* a flock of sheep each having legs, head etc.); they acknowledge the reason:

> "... [this] is not too surprising since wholes [*i.e.* composites] are meant to be more structured than sets."

We take the alternate view that composites have different, less uniform substructure than sets do. Each is incorporated into a uniform representation for substructure embodied in the component relation in ELK. Concerning typing for parts of an entity (*e.g. paw*, *branch*), they say:

> "[this] uncovers an active area of current research, which is beyond the scope of this paper"

We have addressed both of those problems in ELK.

[Kim et al, 1987] describe a syntax and semantics for supporting composite objects in an object-oriented data base system. They say that defining a collection of objects as a composite has advantages. They give a simple characterisation of a composite using BNF notation which has this as one of the cases, but they do not discuss it any further. They seem to recognise that it is a good idea, but don't say anything further; the claim appeared to be unsubstantiated. They do not distinguish between sorts and set types (*e.g.* *wb*, and *set*(*wb*)); nor do they discuss subdivisions.

They make some assumptions that are more strict than ours. For instance, they do not allow the existence of a part without the whole. For example, a car door cannot exist without its owner, a car. Although this may be a reasonable restriction in many cases, it is unacceptable in the ecological modelling domain. A modeller should be able to create a branch without a corresponding tree. Other issues that they are concerned with are:

1. *shared parts*: One way is to copy instead of sharing to achieve the effect; but this makes maintenance a pain. You also have to prevent loops. The difficulties are such that shared sub-parts typically are not allowed. This is

achieved by allowing an object to be made a part of a composite object *only at the time of creation* of that object.

2. *dependent parts*: If you delete an object, do you also delete all it's parts? You might want to attach the part to an as yet uncreated object.
3. *consistency*: Updating a possibly huge data bases of objects when the class taxonomy changes is a difficult problem anyway; adding composite objects just makes matters worse
4. *concurrency*: this was the chief concern in [Kim et al, 1987]

Only the first is relevant to the current implementation of ELK. With respect to shared parts, we have seen that the above approach is not always the right thing to do in the ecological domain. For example, a sheep can be a member of two overlapping flocks and a region may be part of two different regions if they overlap. The second two points are not addressed but could be in the future. The final point is concerned with problems of many people editing a large object base simultaneously and is of no relevance to ELK.

8.6.2.2 Formal Theories

[Hayes, 1986] was one of the first to discuss some of the deep and widespread problems of representing substructure. One interesting example questions the identity of a car if every single part is replaced, one by one. Another challenging concept is to represent various properties of fluids. These two problems are not relevant to ELK.

There is a substantial body of work in the natural language literature concerned with representing plurals and the mass/count distinction. For instance, if you create a whole consisting of three stones, the collective object is a set of three stones. On the other hand, if you create a whole consisting of three quantities of mud, then the collective object is not a set of three muds. Depending on whether they have been mixed, the collective entity is three quantities of mud, or just mud. In the natural language literature, this distinction between types of wholes is called mass versus count. Other terms for mass nouns are 'stuff', 'substance', etc. Various logic formalisms have been created to help make these distinctions.

Because there is never a requirement to represent individuals explicitly, collections in ElkLogic behave like continuous substances (*i.e.* 'stuff') in that there is no limit to the number of subdivisions, sub-subdivisions, etc. that a collection

may have. In ELK, this is captured by the fact that $\#S \subset^p \#S$. On the other hand, the type of the whole consisting of components all of whose sorts are S is <u>not</u> S, but rather $\#S$ (assuming S is non-homogeneous). Just how ElkLogic might be best extended to handle substances has not been explored. To some extent, we can already achieve the desired effect without extending the logic. In particular, we assert $S \prec^p S$ for any sort S which is continuously decomposable (*e.g. region*, *sand*). This in turn implies that $S \subset^p S$ which must not be true for discrete sorts like *sheep*. The advantages and disadvantages of this approach require looking into. An alternative more general approach would be to distinguish between two kinds of sorts mass sorts, and discrete sorts. The former are continuously decomposable (*e.g. region*); the latter are not (*e.g. tree*). We would have to add another case to the definition of $\subset^p_\circ$ such that for all mass sorts S_m, $S_m \subset^p_\circ S_m$. This would complicate the definition of the base cases for the possible component relation ($\subset^p_\circ$); its effects need to be explored. Nothing is implemented with respect to mass sorts at this time.

Logic of Plurals and Mass Terms

One classic work is by Link [Link, 1983]. His logic of plurals and mass terms (LPM) is very much concerned with the substructure of the world. Although designed to capture many subtle distinctions that are not of interest to us, there are important similarities. Collective (*i.e.* plural) objects in ELK are instances of some set type (*e.g.* $\{shp1, shp2\} : set(sheep)$). If this were part of a named flock, then we would have: $shp1 \subset flk$ and $shp2 \subset flk$. In LPM, this would be represented as $flk = shp1 \oplus shp2$. LPM distinguishes between an object and the stuff which constitutes it.

> "...if we have, for instance, two expressions a and b that refer to entities occupying the same place at the same time but have different sets of predicates applying to them, then the entities referred to are simply not the same. From this it follows that my ring and the gold making up my ring are different entities; they are however, connected by what I shall call the *constitution relation* ... [Link, 1983]"

Link also notes the similarity between stuff and collections of things. Both may be decomposed into parts that are exactly the same sort of thing (*e.g.* a region may be composed of regions, a set of sheep may be composed of a set of sheep, but a tree is not composed of trees).

Ensemble Theory

A logical theory called ensemble theory [Bunt, 1986] has been developed that can represent and reason about continuous substances. It allows easy shifting back and forth between a discrete and continuous view of the same substance. Consider sugar. Normally is is continuous, and behaves like a liquid. However, we may wish to view it as a set of individual granules. This is easily handled by this theory.

Ensemble theory has important similarities to ElkLogic, including motivation. According to [Bunt, 1986], it is based on a theory of mereology first developed by Lesniewski between 1911 and 1922 as an alternate to set theory and which incorporated the part-whole relation (or component-whole, as we would call it). Most of the original papers were in Polish, and thus fairly inaccessible. Leanard and Goodman reformulated it into a theory they called the 'Calculus of Individuals' [Leanard & Goodman, 1940]. The notion of individual is introduced by contrasting them with sets or classes (*c.f.* our sort *indiv*).

> The concept of an individual and that of a class may be regarded as different devices for distinguishing one segment of the total universe. ...In both cases, the differentiated segment is potentially divisible, and may even be physically discontinuous. The difference in the concepts lies in this: that *to conceive a segment as an individual offers no suggestion as to what these subdivisions, if any, must be, whereas to conceive a segment as a class imposes a definite schema of subdivision into subclasses and members*[my italics].

Independently, we made the observation in italics. This gives rise to two of the three basic kinds of substructure in ELK and is directly reflected in the definition of the possible component relation. In ELK, the suggestion about the nature of the subdivisions of segments conceived of as individuals rather than sets is expressed in the $\prec^p$ relation (*e.g.* *branch* $\prec^p$ *tree*). Bunt summarises Leanard and Goodman's remarks from that paper:

> The authors argue that discourse can often not be modeled adequately in set-theoretical terms, because set theory has no formal relations for describing the internal structure of individuals: "The ordinary logistic defines no relations between individuals except identity and diversity". A calculus of individuals that introduces other relations such as the part–whole relation would obviously be very convenient.

However, Bunt notes that there are problems with mereology. In particular:

> "The part–whole structure of the individuals has the same logical properties as the part–whole structure of sets as defined by the subset relation. Adding the axioms of the calculus of individuals to those of set theory therefore leads to an axiom system in which a part–whole structure is defined twice: once indirectly, via the axioms for the membership relation, and once directly by the mereological axioms.[4] "

ElkLogic is not rigorously defined as a set of axioms and inference rules, nor has it a formal semantics; thus it is difficult to make concrete comments about where ElkLogic stands with respect to the above quote. However, it is certainly the intention that the single component relation which captures all three cases of substructure is meant to avoid exactly this kind of duplication.

Bunt goes on to define ensemble theory which although specifically designed to handle continuous substances, actually does much more. It encapsulates both set theory and the calculus of individuals as special cases. In this sense, it is fundamentally similar to ElkLogic's representation of substructure. He formally introduces a distinction between continuous and discrete things (ensembles), which can be used to capture the necessary distinction between mass and count nouns.

His ensembles are objects quite like entities in ElkLogic which may or may not have substructure. "Those ensembles that do not have any atomic parts at all [are] used to model continuous substances." There is an analogy here with a collection object as noted above which happens not to be composed of individual sheep. However, this certainly does not capture the sense of continuous substance that is intended. A flock of sheep is certainly not a continuous substance. The most natural way for ElkLogic to be extended is to introduce a new kind of sort for continuous substances.

A desirable feature of such a theory is the ability alternately to view any continuous substance as consisting of any number of kinds of discrete elements depending on their needs. Thus, a person should be able to specify sand as a mass sort without *having* to talk about the grains that compose it; but they should *be able to* if required. Ensemble theory handles this very elegantly.

To some extent, ElkLogic can handle this as well. Some of the properties of continuous sort can be obtained by using the $\prec^p$ relation (we saw this with respect

---

[4] Here 'part' is being used as we use 'component' in ElkLogic.

to *region* in § 6.7.3). Consider *sand* and *grain*. Because sand can be subdivided into two entities both of which are sand, we require $sand \prec^p sand$. The same is not true for grains of sand, so $grain \not\prec^p grain$. Because sand consists of grains, we also need: $grain \prec^p sand$. An entity *snd:sand* might be alternatively viewed simply as sand or if the user desired, it could be viewed as a set of entities of sort *grain* by editing the component relation. For example, $grn_1 \subset snd$, $grn_2 \subset snd$ would specify that there were two particular grains in the quantity of sand denoted by *snd*. We have seen a similar phenomenon with time. For example, we can view *yr* as a year, or as a set of 365 days. Analogously, currently a user has the option to view a tree as a whole, or as a set of branches, a trunk, leaves, etc.

The resemblance between ensemble theory and ElkLogic is strong. One point of similarity is the ability easily to accommodate different viewpoints. In ElkLogic, a whole can be viewed as an individual, or as a set of its components. A sheep can be viewed as a set of organs, as a sheep, or as an animal. In ensemble theory, you can smoothly shift between a continuous and discrete view.

One major difference is that ensemble theory does not use an order sorted logic; thus there is no discussion about an explicit relationship between subsort, part and component. Another major difference is that ensemble theory is a rigorous theory with no implementation. ElkLogic has been implemented, but it is not a rigorous theory.

Bunt would perhaps be surprised that a substantial part of his theory has been implemented. This is because he explicitly follows Pat Hayes' view that it is a better idea *not* to think about implementation issues. Hayes argues that this can overly restrict your imagination ultimately slowing progress in achieving good theories for the common sense world [Hayes, 1986].

Extending Ensemble Theory

There was a paper in [Raulefs, 1987] which extended ensemble theory; it is mostly concerned with continuous substances. He integrates ensemble theory into an object-oriented representation. In particular, their extensions gave rise to:

> ... a mathematical foundation for representing and reasoning about dynamic systems with continuous objects such as liquids, and continuous processes, such as chemical reactions ...

This has obvious relevance for modelling applications. Again, formal rigour has been provided, but no implementation is described.

8.6.3 Summary: Knowledge Representation

We are aware of no implementation other than ELK which has attempted to do the following:

- distinguish sets all of whose elements are the same kind from sets that necessarily contain all elements of a certain kind.
- allow inducing of class variables from instance variables (*e.g.* average height from height)
- allow selective inclusion or ignoring of attributes and parts
- incorporate member and subset as well as part-composite into a uniform framework.

There are a great many problems related to substructure that are of no relevance to ELK. [Kuczora & Cosby, 1989] suggests that there will be no general solution to the substructure problem. Rather, each domain will have special requirements which will be catered for. This is what we have done in ELK.

8.7 Main Contributions of Thesis

Below, we summarise the main contributions of this thesis.

1. *Artificial Intelligence and Simulation*: New approach to representing and constructing ecological simulation models. The main advantages are with respect to:
    - *model comprehension*: an explicit ecological account of the simulation model is represented.
    - *expressive power*: formal representation of a significant body of knowledge about ecology and simulation modelling.
    - *reduced conceptual distance*: users may describe ecological systems and models in familiar terminology
    - *managing choices*: identification and control of idealisation search space

- *consistency checking*: to a large extent users are prevented from describing:
  - ecological systems that do not make sense
  - simulation models that are inconsistent with the ecological system

The main techniques used to facilitate the above are:

- decomposition of the modelling process into three distinct phases:
  1. general/ecological knowledge base
  2. description of ecological system
  3. simulation model
- use of typed lambda calculus; in particular:
  - rich type structure
  - few representation primitives
  - functions used to combine primitives to form complex expressions
  - association of ecological meaning to each primitive and function
- gradual elaboration
  - from vaguely to precisely formulated concepts
  - from ecological to simulation modelling concepts
  - from simple to complex concepts

Contributions to simulation technology, independent of the ecological domain:

- the first extensive effort to assist in building models from scratch
  - in an ill-structured domain
  - where there is no 1-1 correspondence with respect to available tools.
- improved model comprehension in domains where:
  - convenient 1-1 correspondence is not possible
  - explicit representation of idealisations decisions is useful
- high-level modelling language complete with user interface. Significant extension of work in object-oriented representation of differential equation models [Lounamaa, 1986].

2. *Software Engineering*: We have applied many important ideas and techniques in software engineering to simulation modelling, a new software application area. We make no claims about major new techniques. Many of the same techniques are used, although in slightly different ways due to the characteristics of domain. The main areas in software engineering that are relevant to the research described in this thesis are:
   - software comprehension
   - domain analysis
   - software reuse
   - requirement capture
3. *Knowledge Representation*:
   - novel use of type lambda calculus:
     - for defining and controlling idealisation search space rather than to enhance deductive efficiency
   - set substructure
     - new technique for inducing a wide range of useful attributes.
4. *Analysis of Problem of Formalisation*: We defer the details of this until chapter 9.

8.8 Conclusion

This concludes the main portion of the thesis. In the next chapter we summarise the salient aspects of the thesis and repeat the main conclusions that we have already stated in the main body of the thesis. Additionally, we summarise the results of our analysis of the general problem of formalisation.

Chapter 9

Summary and Conclusion

9.1 Summary

Part I: Analysis of Problem

We are chiefly concerned with two important issues in simulation modelling: *model comprehension* and *model construction.* The domain that we are considering is ecological modelling. To gain insights into how we may go about this, we considered formalisation problems in general.

In chapter 1 we introduced the formalisation problem and noted that although many techniques and approaches have been used in many contexts, the inherent difficulties do not seem to go away, rather they resurface in a different form. We stated the following general objectives of this thesis:

1. to identify key issues and difficulties relevant to the formalisation problem in general
2. to relate these to the phenomenon of difficulties not going away, but resurfacing in different forms.
3. to find out to what extent it is possible to stop the infinite regress of reappearing difficulties and produce useful systems which alleviate the formalisation problem.
4. to discover general techniques that may be used to solve the formalisation problem in a variety of contexts.

The remainder of chapter 1 reports on the results of the first two objectives. The bulk of the thesis is concerned with the third objective in the context of ecological modelling. To the end of achieving the fourth objective, we proposed to explore the extent to which identification of the nature and use of *goals* in a formalisation problem may be useful. This is an important theme of this thesis.

9.1.1 The Formalisation Problem

We characterised the general problem of formalisation. Specifically, we noted various difficulties that arose and identified general techniques that may be applied to overcome them. The main difficulties are summarised below.

- *Adequate Formalisms*:
  - syntactic adequacy
  - semantic adequacy
    - expressive power
    - conceptual distance
- *Choice Management*:
  - choice types:
    - *what* to do
    - *how* to do it
  - level of assistance:
    - *identify* range of possibilities
    - *prune* options that do not make sense
    - *advise* on preferred choice

The major techniques that we identified for overcoming these difficulties are:

- choice and design of formal languages
- intelligent and/or user-friendly interfaces
- interpreters/translators
- richly typed languages
- consistency checking

Choice and design of appropriate language is very important. To some extent this may alleviate problems with syntax. More fundamentally (a) languages are

needed to express the required concepts in a domain and (b) the semantics of these languages must be expressed in terms that end users are likely to be familiar with, thus reducing conceptual distance.

Designing languages with sufficiently small conceptual distance frequently means that additional interpreters and/or translators are required to express the final formal representation in some more useful target language. Interpreters may be incorporated into user-interfaces which directly translate into the target language, or a separate translation phase may be required. User interface techniques may also be used to alleviate difficulties with poor syntax. In these cases, the interface effectively redefines the syntax into a new sugar-coated version. Important techniques for managing choices (*i.e.* search space control) are (a) consistency checking and (b) using a rich type structure in the language. This requires considerable domain knowledge.

We noted the general phenomenon of reappearing difficulties. In particular, when applying one or more of the above techniques, the same difficulties tend to resurface in a different form.

This thesis suggests a methodology for addressing formalisation problems based on the idea of applying the above techniques and embodying the solutions in a computer assistant. The design and implementation of the computer assistant gives rise to new difficulties and thus constitutes a reformulation of the original formalisation problem. In the reformulated version, much of the work is done by the system developers so that overall there is a real gain for end users.

The main body of this thesis reports on an exploration of this idea in the context of ecological modelling. We motivate and describe the design and implementation of ELK, a computer assistant for the ecological modelling formalisation problem.

9.1.2 Ecological Modelling

In chapter 2 we introduced ecology and ecological modelling. We first noted various kinds of models and modelling paradigms. This thesis is concerned with differential equation models. We discussed the nature and range of ecological information that ecological modellers are concerned with. The key concepts are *processes, entities, substructure, attributes, and values.* This is the essence of what we called the

PESAV framework. This framework embodies the requirements for overcoming the difficulty of achieving sufficient expressive power.

Two major problems with the current state of the art in tools for ecological modelling are:

1. *Model Comprehension*: Current representations for models do not record important assumptions about how the ecological system was idealised. This severely restricts the use of simulation models. There is no effective means for:
   - analysis of models.
   - use by others either directly, or as a component of another model.
   - modifying the model.
2. *Model Construction*: Models are difficult to construct because tools do not cater for ecologists who lack programming, mathematical, and/or modelling skills. Conceptual distance is unacceptably high because the available tools do not 'understand' ecological concepts.

The root of both problems is the lack of facilities for representing and reasoning about domain knowledge. The key insight that our achievements are based on is to represent both domain information and simulation modelling information separately with explicit links between the two. This facilitates model comprehension directly; it facilitated model construction by reducing conceptual distance and by identifying the modelling search space.

9.1.3 Ecological Modelling Goals

In chapter 3, we explored the nature and potential uses of goals in ecological modelling. The most important observation is that representation and acquisition of ecological modelling goals is inextricably linked with representation and acquisition of ecological and modelling information. The main results of the analysis of how goals may be used are:

- In the early stages of model acquisition, the main use for goals is to coax the user into identifying the important concepts.
- In the middle and later stages of model acquisition, goals may be used to constrain certain idealisation decisions. To do this however, requires an extensive knowledge acquisition exercise.

- There is potential for using the goal ontology as a dialogue graph to drive the early stages in a model consultation.
- Whichever kinds of goals are identified first, ultimately we are forced to consider specific concepts about the ecological system or model.

This led us to conclude that there is a mechanism for providing assistance in the early stages of the ecological modelling process that does not necessarily depend on goals. It is the specific concepts that the goals mention, not the goals themselves that drive the early stages of model acquisition. In ELK, users may say what they are interested in and/or is important for a particular modelling exercise. Interest may be noted directly using one of the commands for noting interest. An alternative, would be to let interest be specified either indirectly and implicitly by specifying goals. However, because initially the effect is the same, there was no need to implement goals in early versions of ELK.

This in no way undermines the usefulness of goals. From a methodological point of view the goals played a major role in identifying the important ecological modelling concepts, thus defining the requirements for expressive power. This is because the goals are necessarily cast in domain terms. We generalise this experience and suggest that any similar exercise in reformulating a formalisation problem by designing a computer assistant may usefully begin by identifying the nature and possible uses of goals in the original formalisation exercise.

From a practical standpoint, goals are also useful. First, in the middle and later stages of the modelling process, it would be useful to have goals around, because they may be used to guide idealisation decision making. Second, users should be able to express goals if they find it convenient to do so. Furthermore it can be more efficient because it eliminates the need to note interest explicitly. These points were discussed more fully in § 3.5.

9.1.4 Design Considerations

In the previous two chapters we considered the issues of model comprehension, expressive power and conceptual distance in the context of ecological modelling. In chapter 4, we completed the requirements analysis for ELK which was begun in chapters 2 and 3. We concentrated on the general issue of managing choices and syntactic adequacy which are important with respect to the interface. In this context, we identified the following additional major requirements:

- consistency checking: ensuring ecological and modelling sense
- flexibility: allowing users to do things in many different ways and to interleave tasks.
- relief from redundant and/or menial tasks: there is a plethora of chores that the user should not have to worry about.

We identified a further need to distinguish between

- general/ecological knowledge: containing universally accepted knowledge
- ecological system description: containing information about a specific (real or hypothetical) ecological system to be modelled.

This serves three major purposes:

- to ensure that the description of ecological system makes sense
- to identify and prune the search space for describing ecological systems
- to facilitate reuse

Within the simulation modelling information level, we also further distinguished between dialogue level information and the runnable model itself. The former consists of three main types. First there are goals; these are reasons for engaging in the modelling exercise. Next, there are interest specifications which indicate the important aspects of the ecological system that should be considered for the model. Finally, there are a range of user-specified defaults to ease the burden on the user for specifying information of the same type over and over.

The benefits of having both kinds of simulation modelling information are many. The runnable-model information is necessary for obvious reasons. The dialogue information facilitates provision of useful assistance during the modelling process. This includes gradual elaboration, reduced conceptual distance, relieving the user of menial tasks etc. Keeping these levels separate is useful primarily to enhance conceptual tidiness, clarify exposition, and simplify the implementation.

Conclusion: Part I

This concluded the first part of the thesis. In it we

- defined the general problem of formalisation
  - identified major difficulties
  - outlined solution approaches to these difficulties

- defined a particular problem of formalisation: ecological modelling
  - characterised the domain
  - identified major difficulties
  - outlined solution approaches to these difficulties
  - explored the nature and use of ecological modelling goals

The most important outcome of the characterisation of ecological modelling is the knowledge ontology summarised below:

1. Ecological information
   (a) general/ecological level
   (b) ecological system level
2. Simulation modelling information
   (a) Dialogue Level
      - goals
      - interest/importance
      - user-specified defaults
   (b) Runnable model level

This gave rise to two major hypotheses of the thesis:

> *That building a computer assistant based on our knowledge ontology can facilitate the achievement of the claimed benefits in the context of ecological modelling.*
>
> *That every piece of information that is deemed to be useful in the process of constructing ecological models can be unambiguously placed into our ontology.*

In the second part of the thesis we investigated these hypotheses by describing the theory and implementation of ELK.

9.1.5 ElkLogic

In chapter 5 we described the theory underlying the representations used in ELK. The language is based on the typed lambda calculus. The following features are of particular significance:

- the representation for sets, in particular the inducing of set types from sorts using the type function *set*.
- representation of substructure
  - incorporates the three different kinds of component whole relationships:
    - member–element
    - subset–set
    - part–composite
  - the use of the induced relation $\subset^p$ to help ensure ecologically consistent specification of substructure relationships
  - the representation of substructure information about instances in their names rather than their (sub)types.
- the use of higher-order functions
  - to represent the concepts of rate, average, maximum etc. These are used to induce attributes of sets of entities of a certain sort based on the attributes of the sort (excluding rate).
  - to map attributes and effects to model variables, allowing idealisation decisions to be identified and recorded

9.1.6 ELK: Representation

In chapter 6 we described the implementation details with respect to the representations used in ELK. Many of the object-level concepts have virtually direct representation as Prolog predicates. There are also many meta-level constructs that are used to define the object-level ones. To investigate the ontology completeness hypothesis, we classified every one of the constructs into one of the four categories in the ontology.

We first described how [pure] model variables and schema are represented. This was followed by the relevant details for representing the general/ecological knowledge base, the description of the ecological system, ecological model variables, and ecological schema. Only ecological variables and schema (not their pure counterparts) encode an explicit representation of their ecological meaning.

Next, we gave the details for how the dialogue level constructs are represented. These include goals, interest specifications, and user-specified defaults.

The remainder of this chapter demonstrated that the representations can be used to satisfy the following three major requirements:

- model comprehension
- expressive power
- reduced conceptual distance

We illustrated the mechanisms we have to explain the simulation model in ecological terms; this facilitates model comprehension. The knowledge ontology plays a crucial role; the key is the separation between the ecological and simulation modelling levels. There are three main techniques that facilitate model comprehension. First we have automatic documentation of ecological model variables. Second we represent some important idealisation decisions explicitly. In particular, we record how attributes and effects are idealised as model variables. Also, using the *ignore* option, ELK can explicitly represent the fact that a user recognises the potential importance of something, but that it will be ignored for the current simulation model. The third way we facilitate model comprehension is by recording a variety of idealisation decisions implicitly. Anything that is in the description of the ecological system, but not in the simulation model constitutes an idealisation decision to ignore that part of the system in the simulation model.

We illustrated the expressive power by showing how the constructs may be used to represent the simple Serengeti model first described in chapter 2. A far richer representation was developed which contained a great deal of ecological information. The main techniques we use to facilitate expressive power are: (a) rich typing, (b) few primitives, each reusable in different contexts and (c) combining functions.

We used two main techniques we used for reducing conceptual distance. First, the semantics of the representation language are expressed in ecological terms. The second technique is gradual elaboration. It is used to go from ecological to

modelling concepts, as well as from simple to complex concepts. Of critical importance here is the four-level knowledge ontology which gives rise to a sequence of constructs which help bridge the gap. Use of combining functions like *maximum*, *average*, etc. allows users to create complex logical terms gradually.

The important role of the knowledge ontology in the facilitation of reduced conceptual distance and model comprehension supports our ontology usefulness hypothesis. Also, the distinctions that give rise to the ontology contribute significantly to the expressive power of the language. Additional support is discussed in chapter 7.

The successful classification of all the constructs in ELK, and virtually all the concepts related to the sample model into our knowledge ontology, supports our ontology completeness hypothesis.

9.1.7 ELK: Elicitation

Chapter 7 described and illustrated how to use ELK. In doing so we addressed the following major issues/requirements that had not been covered yet:

- syntactic adequacy
- choice management
- consistency checking
- flexibility
- relief from redundant tasks

We first gave a general overview of the interface and facilities. This was followed by a step by step illustration of how ELK is used to create formal descriptions of ecological knowledge, systems, and models. There are three distinct modelling phases required for using ELK. Each embodies a considerable degree of idealisation and results in a description in one of our information levels. The phases and intermediate descriptions are summarised as follows:

- real world (not represented!)

  Phase I: [real world] idealised as:
- general/ecological knowledge base

  Phase II: [conceptual model of the world] idealised as:
- ecological system description

  Phase III: [description of the ecological system] idealised as:

- runnable model

Phase I consists of defining the sort and part hierarchies, as well as creating and characterising both attributes and processes. Phase II consists of creating entities (individuals and collections), specifying substructure between these entities, and specifying occurrences of processes. Phase III has two parts. The first consists of specifying goals, interest/importance, and user defined defaults. The second consists of creating model variables (usually by idealising attributes and effects), specifying the computation network (*i.e.* how to compute intermediate and exogenous variables), and initialising state variables and parameters. The differential equations are implicitly defined, the user need not interact with them directly, although they may be displayed.

There are certain inherent restrictions about the order in which users may do things. For example, a user may not create a variable corresponding to the attribute biomass of some sheep entity (phase III) unless the entity has been created (phase II), and the attribute has been created (phase I). Furthermore, the entity cannot be created unless a sort for sheep has been created (phase I). Aside from this, users may do things in any order they like. For example, they may choose to specify most of the ecological system before specifying any of the model. Alternatively, before specifying any other part of the sort hierarchy or ecological system, they may choose to deal with one concept from the creation of a sort, and attributes, to specifying a model variable and how to compute it.

For each category, we showed how users may execute commands to achieve the desired effect. We also noted what support the system offered and how with respect to the major requirements listed above. In chapter 7 we also justified the design rationale for ELK presented in chapter 4. It did so by reconsidering the major requirements which motivated the design and showed how the various techniques were used to meet these requirements. This is summarised below from a techniques point of view, contrasting from the requirements oriented presentation in the rest of the thesis (especially chapter 4).

rich type structure: facilitates the requirement for relatively few primitives; facilitates consistency checking, in conjunction with identification and pruning of the specification search space; used to implement the notion of permission for consistency checking.

few primitives: in conjunction with the type structure and various combining functions facilitates retaining explicit connections; in conjunction with the typing and combining functions facilitates implicit specification

combining functions: each with an explicit ecological meaning facilitates automatic documentation; also facilitates identification of specification search space and gradual elaboration

implicit specification: in conjunction with few primitives, and various combining functions facilitates no redundancy, space efficiency, and modifiability

retain explicit connections: this is the key to expressive power; facilitates provision of general purpose automatic documentation mechanisms to produce English descriptions of the state of the specification in ecological terminology

automatic documentation: facilitates model comprehension, and transparent interface

knowledge ontology: facilitates identification of idealisation search spaces, model comprehension, consistency checking, relief from redundant/menial tasks, gradual elaboration, and reuse; also the distinctions give expressive power

gradual elaboration: facilitates reduction of conceptual distance;

inheritance and defaults : facilitates relief from redundant/menial tasks

user-driven dialogue: facilitates flexibility.

Conclusion: Part II

This concluded the second part of the thesis whose main goals were to

- describe the theory and implementation of ELK
- investigate the ontology usefulness and completeness hypotheses
- justify the design rationale for ELK described in chapter 4

We say no more about the first point. The usefulness of the four level knowledge ontology that underlies the theory and implementation of ELK is evident from the above discussion associating techniques and requirements. Most of the techniques support up to a few of the major requirements, and some minor ones. The knowledge ontology, on the other hand *directly supports all of the major requirements* (except flexibility). It also supports various important but less fundamental requirements.

The following evidence supports the knowledge completeness hypothesis:

- the attempt to unambiguously classify all the constructs in the language into one of the four levels was successful
- the attempt to classify all the relevant information with respect to the example model was largely successful.

The support for this hypothesis is weaker than that for the usefulness hypothesis. More models need to be examined. However, discovering whether the hypothesis holds is not the main objective. The point of identifying and investigating the hypothesis is to increase understanding of the modelling process. That the distinctions in the ontology are useful is clear. If it turns out that the completeness hypothesis is true, so be it. If it does not, then the process of analysing situations where the knowledge ontology breaks down is likely to lead to the increased understanding of the modelling process that we seek.

The design rationale for ELK is summarised in figure 4–3. The major requirements constraining the design for ELK were (in approximate order of decreasing importance):

- model comprehension
- expressive power
- conceptual distance
- choice management
- consistency checking
- flexibility
- relief from redundant tasks

The major techniques that support one or more of these requirements are:

- rich typing
- few representation primitives
- use of functions for combining primitives to form complex expressions
- retain explicit connections
- distinct information/knowledge levels
  - ecological
    - general/ecological
    - ecological system
  - simulation modelling
    - dialogue

- runnable model

- gradual elaboration
- automatic documentation
- implicit specification
- defaults
- user-driven dialogue

The fundamental techniques that support virtually all the requirements discussed in chapter 4 are:

1. the rich typing
2. relatively few ecologically meaningful primitives in conjunction with a variety of ecologically meaningful combining functions. The most important by far of the latter is *set*.
3. the four level knowledge ontology

The nature of the constructs which give us the required expressive power is the foundation on which our techniques for meeting the remaining requirements rest. That is, the constructs have been carefully designed to support (1) the development of easily used interface facilities, (2) reducing conceptual distance, and (3) managing choices. The fact that the semantics of the constructs are expressed in terms that users are familiar with supports the first two. The four-level knowledge ontology is the essence of our gap-bridging technique. The support for the choice management derives from the rich typing, in conjunction with the knowledge ontology. This is manifest in the identification and control of the elicitation search space which simultaneously supports a wide variety of consistency checking. The idea is that by encoding sufficient knowledge in the domain we can prune the search space by ensuring consistency.

Part III: Discussion

9.1.8 Related Work

In chapter 8, we compared the methods used by ELK with those of other projects and systems which share similar goals. The key areas were:

1. Ecological Modelling

2. AI and Simulation
3. Software Engineering
4. Intelligent Human Computer Interfaces
5. Knowledge Representation

With respect to 1 and 2 the primary issues we have addressed are comprehension, and construction of simulation models. The key to our solution is representing separately domain knowledge and the simulation model with links between them. We have made substantial contributions to these fields.

With respect to 3, our work is relevant to the subfields of requirements capture, reuse, domain modelling, and software comprehension. There is considerable overlap in techniques used, however applying them in the software engineering domain of simulation modelling is new.

With respect to 4, the issues that we have addressed include reducing conceptual distance, relief from redundant/menial tasks, and flexibility.

With respect to knowledge representation, we invented a variety of domain independent techniques to meet difficult challenges in the domain of ecological modelling. Of particular interest are sets and substructure. From a practical standpoint, our representation had to support the ability to ignore attributes and parts that necessarily apply to things, but which might not be relevant to a particular modelling exercise. Although some tools may allow some of these facilities to be added, we are aware of no currently available tools that support these requirements directly.

9.2 Conclusion

9.2.1 The Problem of Formalisation

The key difficulties in a typical formalisation problem are: syntactic adequacy, expressive power, conceptual distance, and choice management. One way to alleviate these difficulties is to build a computer assistant. This will require a considerable amount of domain knowledge. The design and implementation of such an assistant constitutes reformulating the original formalisation problem into a new, hopefully easier to solve one. The need for a computer assistant gives rise to the following requirements: consistency checking, flexibility, and relief from redundant/menial tasks.

We propose the following domain-independent guidelines/methodology for constructing a computer assistant to alleviate the above difficulties and meet the above requirements.

1. Identify the nature and use of goals in the domain of interest.
   - initially to determine the expressive power requirements
   - subsequently to provide advice on 'good' idealisation decisions
2. Choose/design a representation language which meets the requirements of expressive power and small conceptual distance. In particular:
   - adequate coverage of domain of interest
   - semantics are cast in terms that end users understand
   - difficulties with providing user interface should be minimised
3. Use a rich type structure in the language; this facilitates choice management and consistency checking.
4. Use the syntax and semantics to generate and prune the search space that is implicit in the language. This
   - requires domain knowledge.
   - maintains consistency.
5. Identify a corpus of domain specific heuristics for advising on how to make good decisions in the formalisation process.
6. Build an interface which incorporates the above and ensures syntactic adequacy (via 'sugar-coating').

We have illustrated that this methodology works in the domain of ecological modelling. This is embodied in ELK. The least amount of effort was devoted to step 5. The are two main reasons for this:

1. there were many issues which needed to be addressed first
2. in the ecological modelling domain, this knowledge is not readily available; moreover there is not much general agreement

The first reason is likely to apply to a wide range of formalisation problems. Depending on the nature and difficulty of the task, the need for advising on making good decisions may vary in extent. In the ecological modelling domain, its general lack means that the system will not be as easy for inexperienced modellers to use. However, experienced modellers may well ignore whatever advice the system gave and do what they wanted to. For them, such advice is at best a convenience and a double check; at worst, it is unnecessary.

Ideally, there will be a readily available language or framework which can be used or modified. It is especially convenient if the language provides a one to one correspondence with respect to the terms that users think in (*e.g.* ECO and KBS) because then the problem of conceptual distance does not arise. Otherwise it may be necessary to design a sequence of constructs in the language, or having separate levels (*i.e.* sub-languages) with explicit relationships between each level. This is illustrated by the four-level knowledge ontology in ELK.

In the domain of software engineering, an additional fundamental requirement arises: *software comprehension.* To facilitate this, we propose that at least two distinct layers in the representation language are required. One describes the domain in which the software applies, the other is the (possibly runnable) specification of the software itself (*e.g.* LaSSie [Devanbu et al, 1990]). There must also be formal links between the two layers.

In the domain of simulation modelling, we call this requirement *model comprehension.* The most important information that must be recorded is;

- a description of the system being modelled
- the explicit relationship between the model variables and the system being modelled
- the modelling decisions

We further suggest that the component of the language for describing the system being modelled be divided into a domain layer and a system layer. The former is a repository for general and specific knowledge about the domain. The latter is a specific description of the particular system of interest. Designing representations in this way facilitates the achievement of several benefits simultaneously:

- model comprehension
- identification and pruning of idealisation search space
- consistency checking
- reuse

9.2.2 Contributions

9.2.2.1 Simulation Modelling

The most significant and immediate benefits of this research are likely to be in the area of simulation modelling. The details of what we have accomplished will be more directly useful in ecological domains, however most of our techniques are more general. The most important advances are in the areas of:

1. model comprehension
2. model construction (via computer-aided assistant)

These two areas are intimately connected. First, the basis for solving *both* problems is the same; *i.e.* the development of a language for representing the distinction between ecological and simulation modelling information, and forming a bridge between the two. Second, and more generally, model comprehension itself directly contributes to making models easier to construct.

The most original contribution is in the area of model comprehension. Analogous work has been done in the sub-field of software engineering as discussed in [Soloway & Ehrlich, 1984], however we believe we are the first to address the problem in the domain of simulation modelling.

We have also made a significant advance by developing a new way to construct simulation models based on controlling the vast modelling search space. The usual search space for constructing simulation models is implicitly characterised by the simulation language being used. For general languages (*e.g.* Fortran) this search space is uncontrolled and wholly disassociated from the domain of interest. Thus all the major difficulties for formalisation problems are present:

uncontrolled choices, insufficient expressive power and conceptual distance is very high. Some specialised languages exist which for certain classes of models reduce the choices, and reduce conceptual distance. Such tools are often easy to use, but only in their limited range of applicability. We have opted for an approach which covers a wider range of models, but also keeps conceptual distance small and offers significant choice management facilities. This is done chiefly through the use of domain information.

We are aware of no tools that offer facilities for representing or reasoning about domain concepts *independently* from the simulation model or models that ultimately are created. Instead, simulation models are specified or created directly. This means that there is no way to control the myriad of important idealisation choices that arise in any modelling exercise; or to record the decisions. By contrast, ELK is used to build up a sequence of models, each being used to identify and constrain the important idealisation decisions for the next one. Users first describe general domain information; this is used to describe a particular situation in that domain; finally a simulation model of that situation is constructed. This achieves the goal of extending the range of models expressible compared to specialised systems, but also keeps conceptual distance small and helps control choices.

We have demonstrated that it is possible to construct models using this method in the domain of ecology. However, ELK has not been extensively tested. We are optimistic that ecologists will be able to use this new model construction methodology to construct simulation models more easily that was previously possible. We now review the key techniques that underpin our achievements in *both* of the key areas of model construction and model comprehension *simultaneously.*

The single most fundamental and important idea is to distinguish between four separate layers of information (*i.e.* the knowledge ontology). With respect to the model comprehension, these distinctions facilitate automatic documentation both by explaining the simulation model in terms of the ecological system, and by recording idealisation decisions. This ensures that models are comprehendable which in turn facilitates reuse and modifiability. With respect to model construction, the distinctions in the knowledge ontology facilitate reduced conceptual distance, managing choices and consistency checking.

The next most important idea is our use of logic as a representation language. By itself this is not original, however there are a number of features (some novel) in ElkLogic that are particularly well-suited to our needs. These are:

1. the rich type structure
2. few primitives in conjunction with a variety of combining functions (especially higher-order functions)
3. each construct has a semantics
4. the rich representation of sets and substructure.

Together, these facilitate three key objectives. First, general mechanisms can be constructed which generate meaningful English explanations for a wide range of information from the domain level to the simulation modeling level and the bridge linking the two. Higher order functions play a key role in making the link and keeping the number of primitives down. Next we obtain considerable expressive power, able to cope with a wide variety of complex concepts in the domain of ecological modelling. Thirdly, this defines a substantial portion of the ecological modelling search space. This is the key to managing the myriad of choices that face a modeller. Globally, 1 and 2 facilitate gradual elaboration by allowing the nesting of various functions. Locally, 1, 2 and 3 provide fundamental support for menu management. Every time a user is faced with a choice, menus can be made available which identify the options. Usually, the set of options are dynamically generated. The key idea here is the use of the type system and semantic information to simultaneously identify and prune the search space as well as ensure consistency.

The key to the representation of sets and substructure is the type function *set*. This is used for:

- representing sets that are not types
  e.g. $flk{:}set(sheep)$ versus $sheep$
- inducing attributes for sets
  e.g. $qnam\_ent(maximum, weight) : set(phys\_obj) \times time \mapsto positive.$
- uniform representation of three kinds of component–whole relation

Another unique feature of the logic is representing substructure information in instance names *e.g.* $att\_dim\_fn(flk, age)(old)$

Summarising, the chief contributions of this research to the field of simulation modelling are in the areas of model comprehension and construction . The extent of ELK's competence in both of these areas is significantly greater than that of any current simulation tool that we are aware of. Currently, our framework has been extensively and successfully tested on one real example. More testing is required on further examples in ecology and other domains to prove that our framework works in general.

9.2.2.2 The Formalisation Problem

A less immediate, but potentially significant contribution of this research is in the general area of understanding the formalisation problem. The main achievements in this thesis in this regard are:

- We identified some major difficulties in formalisation problems.
- We identified current approaches for overcoming each these difficulties, both generally, and in the specific context of building a computer assistant.
- We noted the fundamental problem that these difficulties tend to resurface in a different form when solutions are attempted in the context of building a computer assistant.
- We proposed a general framework for reformulating the formalisation problem in such a way as to ensure that the new difficulties that arise are easier to overcome.
- We demonstrated that this framework works in the context of building a computer assistant for ecological modelling.

The extent to which this general framework will prove useful in other formalisation problems remains to be seen. Although developed in the context of ecological modelling, this framework and the majority of the techniques developed in this thesis are independent from the domain of ecology. We are thus optimistic that they will be more widely applicable. It will not become clear until and unless attempts are made to test this framework in other areas. These may include both other modelling domains (*e.g.* economics), and in fields other than modelling (*e.g.* software engineering).

9.3 Future Work

There are a number of ways that work on this project may continue. These may broadly be characterised into the following areas.

1. Tidying up the implementation
2. Extending ELK to include features already designed and/or discussed in this thesis.
3. Empirical testing with various users
4. New features
   - representing a wider range of ecological and modelling concepts
   - subsystem for acquiring ecological schema from users
5. To test the overall approach on domains different from ecology.

Tidying Up and Extending Elk

By 'tidying up', we mean performing conceptually trivial (though often time-consuming) jobs to fill gaps in the implementation. Here, we indicate where the key gaps occur. Full text is not provided for many error, warning, and ok messages. Automatic documentation facilities are in place for many things, but not all. For example, we explain the ecological meaning of model variables, but we have not yet produced text summarising the full range of implicit and explicit idealisations. The information is readily available, it is a simple matter of writing the code to generate the text. The set of available commands, while fairly extensive, is incomplete in some cases. For example, for some constructs, we have not yet implemented commands for modifying or removing them. We have not yet incorporated the process related interest constructs. Finally, indirect reference to value spaces when creating/modifying attributes is not fully implemented.

There are other important jobs that are not so trivial; these fall under category 2. One significant job, although conceptually straightforward is to fully integrate the old goal system with ELK. Included in this would be the addition of explicit constructs for expressing influences and variable dependency. Then, goals may be used to implicitly create both interest *and* influence specifications. Alternatively, as interest specifications may be specified directly, so should influence and variable dependency specification. Influence constructs are idealised as vari-

able dependency which can be used to constrain search for appropriate schemata. Note that influences are at the ecological system level, and express what is true in the ecological system. They may or may not be explicitly represented in the simulation model. If it is, it will be manifest as a link in the computational network. This further extends the bridge over the conceptual gap. Moving further still, to more general and vague concepts involves a more elaborate extension of the goal handling component of ELK. In particular, we can implement the goal ontology as a dialogue graph to allow users to express high-level goals early, and gradually be led to specifying low-level goals (this is outlined in § 3.3.3.1).

Two very time-consuming jobs are consistency checking and recovery mechanisms. There is currently an extensive amount of consistency checking done, and relatively little by way of recovery mechanisms. In the text, we have noted many occasions where it would be possible to add more of this to ELK. Due to the rich type structure and the [informal] semantics, the scope for consistency checking and provision of recovery mechanisms is extensive. Our experience suggests that of all the consistency checking that is theoretically possible, we have probably identified less than 50%. We have only begun to identify the many possible recovery mechanisms that are possible. Very few were implemented. In the short term, the easiest way to identify more will be through extended use of ELK. Other important tasks in category 2 include:

- Automatic creation of model variables when a user specifies *auto* in the *att_inh* construct.
- The saving and retrieving of different portions of the specification in accordance with the knowledge ontology.
- Identifying a corpus of heuristic knowledge to provide assistance on how and when users should proceed in describing ecological systems and models.
- Incorporation of above into a heuristically controlled agenda / suggestion box mechanism which advises the user on what to do next and how to do it
- Macro operations to allow multiple assignments of features and specifications to selected members of a set or parts of a composite.

Some of these are substantial projects in their own right. We say no more here.

Empirical Testing

Ideally ecologists with minimal computing and modelling experience will be able to build, run, and modify complex simulation models based on differential or difference equations models. This needs to be properly tested. Conducting an empirical test would almost certainly yield important information which could be used to guide further development. Some questions that we wish to answer are:

- How much time does it take for users to become familiar with the general facilities provided by ELK?
- Can the many distinctions be safely ignored initially, and gradually learned, (as intended) or are they confusing and burdensome.
- To what extent does the user's background affect their ability to use ELK? Relevant factors include:
  - computer literacy
  - familiarity with object-oriented representations
  - experience with simulation modelling
    - difference/differential equations models
    - system dynamics models
  - degree of expertise in specialised subfield within ecology (*e.g.* population dynamics, forestry)
- Is the conceptual distance sufficiently small? In particular,
  - is the PESAV framework natural or forced?
  - does ELK assist in the making vague information precise?
- Are the facilities for automatic documentation understandable?
- Do users find it useful to express goals first?
- How useful are the consistency checking facilities? Are they overly constraining?
- How much need is there for a system-driven mode which allows users to react rather than initiate.

New Features

Representation We have noted various shortcomings with the current representation used in ELK. For example, the representation for processes is rudimentary. There is much ecological information that it cannot make use of. Also, there

are some problems with the theory of substructure. The notion of homogeneity is relevant to the mass/count distinction in natural language research. This distinction could be used in conjunction with existing techniques in natural language to generate high quality text explaining various aspects of the ecological system and model. The representation might also be extended to include information about the number of parts (*e.g.* one head, four legs). We might replace $\prec^p$ with *part_def* with arguments to indicate the minimum and maximum number of parts (*e.g.* $part\_def(dog, leg, 4, 4)$, $part\_def(tree, branch, 0, \infty)$. We could have a *part_inh* construct which indicated by default how many parts of a certain type an entity may or should have. An *auto* specification might cause these parts to be automatically created, in the same way that variables may be automatically created.

Other shortcomings with the representation will no doubt be uncovered as the range of models that ELK is used to represent expands.

Another useful exercise would be to carry out a more formal analysis of Elk-Logic and compare it with other formal system which address similar issues (*e.g.* ensemble theory [Bunt, 1986]).

Schema Acquisition Subsystem Currently, the ecological schema must be specified manually by the system designers. An interesting and useful project would be to build a subsystem that enabled users to interactively create and modify ecological schema on their own. Considerable effort would be required, as these schema use special purpose Prolog code.

Other Domains

We have already discussed a project similar to this one in the domain of planetary atmospheric modelling [Keller et al, 1990]. It would be very useful to find out the extent to which our techniques apply. Another potentially fruitful domain is that of econometric modelling.

THE END

References

Abrett, Glenn and Burstein, Mark. (1987). *The BBN Knowledge Acquisition Project: Phase 1 - Final Report.* Technical Report 6543, BBN Laboratories Incorporated.

Barendregt, H.P. (1985). *The Lambda Calculus.* Elsevier.

Barstow, D., Duffy, R., Smoliar, S., and Vestal, S. August 1982. An overview of phinix. In *National Conference on Artificial Intelligence*, AAAI, Pittsburgh, Pennsylvania.

Bennet, J.S. and Englemore, Robert. (1979). Sacon: a knowledge-based consultant for structural analysis. In *Proceedings of the sixth IJCAI*, pages 47–49, International Joint Conference on Artificial Intelligence, Tokyo, Japan.

Berman, M. (1979). Simulation, data analysis and modelling with the saam computer program. In Matis, J., Patten, B.C., and White, G.C., (eds.), *Compartmental Analysis of Ecosystem Models*, pages 125–130, International Cooperative Publishing House; Fairland, Maryland, USA.

Biggerstaff, T.J. (1989). Using domain models to understand programs. In *Proceedings of the OOPSLA workshop on Domain Modeling in Software Engineering*, pages 27–31.

Blake, E. and Cook, S. (1987). On including part hierarchies in object-oriented languages, with an implementation in smalltalk. In *European Conference on Object-Oriented Programming*, pages 41–50, Springer-Verlag.

Bledsoe. (1976). *Systems Analysis and Simulation in Ecology*, 4:283–298.

Bobrow, D. and Stefik, M. (1981). *The LOOPS Manual.* Technical Report KB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Center.

Buchanan and Shortliffe. (1985). *Rule-Based Expert Systems (The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley.

Bundy, A. and Uschold, M.. (1989). *The Use of Typed Lambda Calculus for Requirements Capture in the Domain of Ecological Modelling.* Research Paper 446, Dept. of Artificial Intelligence, Edinburgh, Submitted to Logic and Computation.

Bundy, A. July 1984. *Intelligent front ends.* Research Paper 227, Dept. of Artificial Intelligence, Edinburgh, First appeared in the Infotech Pergamon State of the Art Report on Expert Systems.

Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R., and Palmer, M. (1979). Solving mechanics problems using meta-level inference. In Buchanan, B.G., (ed.), *Proceedings of IJCAI-79*, pages 1017–1027, International Joint Conference on Artificial Intelligence, Reprinted in 'Expert Systems in the microelectronic age' ed. Michie, D., pp. 50-64, Edinburgh University Press, 1979. Also available from Edinburgh as DAI Research Paper No. 112.

Bunt. (1986). The formal representation of (quasi-) continuous concepts. In Hobbs, (ed.), *Formal Theories of the Commonsense World*, Norwood Ablex.

Cardelli, L. (1988). Structural subtyping and the notion of power type. *ACM Proc. POPL.*

Cardelli, L. The quest language and system. August 1989. Tracking Draft.

Clayton, B. *Programming Primer: ART (Version 1.1).* Los Angeles, CA, september 1984.

Cohn, A. (1985). On the solution of shubert's steamroller in many sorted logic. In *Proceedings of IJCAI-85*, International Joint Conference on Artificial Intelligence.

Colomb, R., Hearn, B., Brook, K., Jansen, B., Ashburner, N., and Clarke, M. (1988). *Siratac: Expert Decision Support for Cotton Pest Management.* Technical Report TR-FD-88-01, CSIRO.

DeBellis, M. (1989). The corporate domain model. In *Proceedings of the Domain Modelling Workshop*, pages 48–55, Held in conjunction with OOOPSLA-89.

Devanbu, P., Brachman, R.J., Selfridge, P.G., and Ballard, B.W. April 1990. Lassie: a knowledge-based software information system. In *Proc. 12th International Conference on Software Engineering*, Nice, France.

Fickas, S. April 1987. Automating analysis: an example. In *4th International Workshop on Software Specification and Design*, IEEE Computer Society.

Folse, L.J., Packard, J.M., and Grant, W.E. (1989). AI modelling of animal movements in a heterogenous habitat. *Ecological Modelling*, 46:57–72.

Fox, M. (1986). Knowledge-based simulation: an artificial intelligence approach to system modeling and automating the simulation life cycle. In Widman, L., Loparo, K., and N., Nielsen, (eds.), *Artificial Intelligence, Simulation, and Modeling*, pages 447–486, Wiley.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the Language and its Implementation.* Addison-Wesley.

Haggith, M. (1990). Interactive program construction. In *UK-IT 1990 Conference*, pages 227–233.

Hanks, P., (ed.). (1980). *Collins Dictionary of the English Language.* Collins.

Hayes, P. (1986). The second naive physics manifesto. In Hobbs, (ed.), *Formal Theories of the Commonsense World*, Norwood Ablex.

Hayward, S. A. (1988). Structured analysis of knowledge. In Boose, J. and Gaines, B., (eds.), *Knowledge Acquisition Tools for Expert Systems*, pages 149–160, Academic Press.

Hilborn, R. and Sinclair, A.R.E. (1984). A simulation of the wildebeest population, other ungulates, and their predators. In Sinclair, A.R.E. and Norton-Griffiths, M., (eds.), *Serengeti: Dynamics of an Ecosystem*, Universit of Chicago Press.

Holt, J., Cook, A.G., Perfect, T.J., and Norton, G.A. (1987). Simulation analysis of brown planthopper (Nila parvata lutens); population dynamics on rice in the philippine. *Journal of Applied Ecology*, 24:81–102.

Jang, Yeona. (1988). *KOLA: Knowledge Organization LAnguage.* Technical Report 545, MIT.

KEE: Software Development System User's Manual. Mountain View, CA, July 1986.

Keller, R., Sims, M., Podolak, E., McKay, C., and Thompson, D. (1990). *Proposal for Constructing and Advanced Software Tool for Planetary Atmospheric Modeling.* Technical Report RIA-90-03-20-1, NASA Ames Research Center.

Kim, W., Banergie, J., Chou, H., Garza, J.F., and Woelke, D. (1987). Composite object support in an object-oriented database system. In *OOPSLA 87, Orlando*, pages 118–125.

Knowledge Craft CRL Technical Manual. Carnegie Group, (1988).

Kuczora, P.W. and Cosby, S.J. (1989). Implementation of meronymic (part-whole) inheritance for semantic networks. *Knowledge-Based Systems*, 2(4):219–227.

Kuipers, B. (1986). Qualitative simulation. *Artificial Intelligence*, 29:289–338.

Leanard and Goodman. (1940). The calculus of individuals and its uses. *Journal of Symbolic Logic*, 5:45–55.

Legovic, Tarzan. (1987). A recent increase in jellyfish populations: a predator-prey model and its implications. *Ecological Modelling*, 38:243–256.

Letovsky, S. (1986). Cognitive processes in program comprehension. In Soloway, E. and Iyengar, S., (eds.), *Proceedings of the Second Workshop on Empirical Studies of Programmers*, Ablex Publishers.

Lewis, J. (1986). Stella: a model of its kind. *Pract. Comput.*, 9(9):66–67.

Link, Godehard. (1983). The logical analysis of plurals and mass terms: a lattice theoretic approach. In Bauerle, von Stechow Schwarze, (ed.), *Meaning, Use and Interpretation of Language*, pages 302–323, Gruyter.

Loehle, C. (1987). Applying artificial intelligence techniques to ecological modelling. *Ecol. Modelling*, 38:191–212.

London, P. and M., Feather. (1986). Implementing specification freedoms. In Rich, C. and Waters, R., (eds.), *Artificial Intelligence and Software Engineering*, pages 285–306, Morgan Kaufmann.

Lounamaa, P. (1986). An incremental object-oriented language for continuous simulation models. In Widman, L., Loparo, K., and N., Nielsen, (eds.), *Artificial Intelligence, Simulation, and Modeling*, pages 397–413, Wiley.

Lubars, M.D. and Harandi, M.T. (1988). *Addressing Software Reuse Through Knowledge-based Design.* Technical Report STP-058-88, MCC.

Martin, Paul, Appelt, Douglas, and Pereira, Fernando. (1983). Transportability and generality in a natural-language interface system. In *Proceedings of the eighth IJCAI*, pages 573–581, International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany.

Mellish, C.S. (1988). Implementing systemic classification by unification. *Computational Linguistics*, 14(1).

Mitchell, Kenneth J. (1975). Dynamics and simulated yield of douglas-fir. *Forest Science (supplement)*, 21(4):1–39.

Muetzelfeldt, R., Robertson, D., Bundy, A., and Uschold, M. (1989). The use of prolog for improving the rigour and accessibility of ecological modelling. *Ecol. Modelling*, 46:9–34.

Neighbors, J. (1986). The draco approach to constructing software from reusable components. In Rich, C. and Waters, R., (eds.), *Artificial Intelligence and Software Engineering*, pages 525–536, Morgan Kaufmann.

O'Keefe, R. (1985). *Logic and Lattices for a Statistics Advisor.* PhD thesis, Dept. of Artificial Intelligence, Edinburgh.

Patel-Schneider, P. (1984). Small can be beautiful in knowledge representation. In *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, IEEE.

Patel-Schneider, P.F., Owsnicki-Klewe, B., Kobsa, A., Guarino, N., MacGregor, R., Mark, W.S., McGuinness, D., Nebel, B., Schmiedel, A., and Yen, J. (1990). Term subsumption languages in knowledge representation. *AI Magazine*, 11(2):16–22, Report of Workshop held in October 89.

Polya, G. (1945). *How to solve it.* Princeton University Press.

Raulefs. (1987). A representation framework for continuous dynamic systems. In *Proceedings of the tenth IJCAI*, International Joint Conference on Artificial Intelligence.

Reubenstein, H. B. and R., Waters. (1989). The requirements apprentice: an initial scenario. In *Proc. 5th International Workshop on Software Specification and Design*, pages 211–218.

Rissland, Edwina. (1984). Ingredients of intelligent user interfaces. *Int. Journal Man-Machine Studies*, 21:377–388.

Robertson, D., Bundy, A., Uschold, M., and Muetzelfeldt, R. (1987). *Synthesis of Simulation Models from High Level Specifications.* Research Paper RP-313, Department of Artificial Intelligence, University of Edinburgh.

Robertson, D., Bundy, A., Uschold, M., and Muetzelfeldt, R. (1988). *The Eco-Logic System.* Technical Report 1, Department of Artificial Intelligence, University of Edinburgh.

Robertson, D., Uschold, M., Bundy, A., and Muetzelfeldt, R. (1988). The eco program construction system: ways of increasing its representational power and their effects on the user interface. *International Journal of Man Machine Studies*, 31:1–26.

Robertson, D., Bundy, A., Muetzelfeldt, R., Uschold, M., and Haggith, M. (1991). *Eco-Logic: Logic-based approaches to Ecological Modelling.* MIT Press.

Ross, P, Jones, J., and Millington, M. (1985). *User Modelling in Command Driven Systems.* Technical Report RP-264, Dept. of Artificial Intelligence, Edinburgh.

Rothenburg, J. (1989). The nature of modelling. In Widman, L., Loparo, K., and N., Nielsen, (eds.), *Artificial Intelligence, Simulation, and Modeling*, pages 75–92, Wiley.

Saarenmaa, H., Stone, N.D., Folse, L.J., Packard, J.M., Grant, W.E., Makla, M.E., and Coulson, R.N. (1988). An artificial intelligence modelling approach to simulating animal/habitat interactions. *Ecological Modelling*, 44:125–141.

Sinclair, A.R.E. and Norton-Griffiths, M. (1984). *Serengeti: Dynamics of an Ecosystem.* Universit of Chicago Press.

Soloway, E. and Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5).

Stefik, M. and Bobrow, D.G. (1985). Object-oriented programming: themes and variations. *The AI Magizine*, 6(4):40–62.

Thomson, A. and Taylor, C. (1990). An expert system for diagnosis and treatmens of nutrient deficiencies of sitka spruce in great britain. *AI Applications*, 4(1):44–52.

Uschold, M., Harding, N., Muetzelfeldt, R., and Bundy, A. (1986). An intelligent front end for ecological modelling. In O'Shea, T., (ed.), *Advances in Artificial Intelligence*, North Holland, Also in Proceedings of ECAI-84, and available from Edinburgh University as Research Paper 223.

Uschold, M., Robertson, D., Bundy, A., and Muetzelfeldt, R. (1989). Helping inexperienced users to construct simulation programs: an overview of the ECO project. In Vadera, S., (ed.), *Expert System Applications*, pages 117–132, Sigma Press, This is a revised and extended version of a paper published in 'Research and Development in Expert Systems 4', BCSES 1987; Also available as D.A.I. Research paper 338.

Weiss, S., Kulikowski, C., Apte, C., Uschold, M., Patchett, J., Brigham, R., and Spitzer, B. (1982). Building expert systems for controlling complex programs. In *Proceedings of AAAI-82*, pages 322–326, American Association for Artificial Intelligence.

Widman, L., Loparo, K., and N., Nielsen. (1989). *Artificial Intelligence, Simulation, and Modeling.* Wiley.

Wolfe, I.R., Zweig, R.D., and Engstrom, D.G. (1986). A computer simulation model of the solar-algae pond ecosystem. *Ecol. Modelling*, 34:1–59.

Appendix A

Glossary

abstract type: a simplified version of the full type expressed using meta-types.
e.g. $attvar\_fn : attribute \times entity \mapsto propvar$

attribute description: the part of attribute constructs which describes rather than uniquely identify the attribute. This includes: value space, dimension value space, default value, etc.

attribute variable: A variable that represents an attribute of some ecological entity.

attribute-variable conflation: using the same representation for the model variables and attributes. This fails to distinguish the description of the ecological system from the simulation model of it.

benefit: any aspect of our computer assistant that is deemed to be desirable. These will variously be most naturally viewed as issues, requirements, features, or advantages of the system.

choice management: assisting in how or what to do by identifying choices, pruning choices, and or advising on making good choices. This is a fundamental difficulty in the formalisation process.

collection: any entity whose type is a collection type.
e.g. $\{shp, flk\}$:$\#sheep$ where shp:$sheep$ and flk:$set(sheep)$

collection type: any type defined using the # construct. These may also be defined using the *set* and/or $\sqsubset^s$ constructs.
e.g. $\#sheep$, $sheep \sqcup set(sheep)$ constructs.

complete specification: a specification which is such that no further instantiated constructs are required (either syntactically or semantically).

computer model: the embodiment of one or more conceptual models in a formal representation intended for computer processing.

conceptual distance: the degree to which the semantic concepts of a formalism or user interface differ from those that the user is thinking in

conceptual model: a representation of one or more concepts and/or processes which may or may not be manifest in a computer program

conceptual modelling framework: an ontology on which a language for defining conceptual models is based, or such a language.

construct: in our context this will usually be some term with a functor and several arguments. The functor is a primitive in the formalism. It is part of the definition of the syntax of a formalism.
e.g. $component(Component, Whole)$

correspondence: the degree to which the semantic concepts of a formalism or user interface match those that the user is thinking in (the inverse of conceptual distance).

dialogue level: information concerned with the process of constructing a simulation model, but not needed to represent the runnable model; a sub-level of the simulation modelling level distinguished from runnable model level.

domain model: software engineering term for a conceptual model of a software application domain.

ecological attribute: an attribute that may give rise to model variables or be used to define substructure for ecological entities.

ecological entity: entities that have ecological attributes; this is meant to include all things that may be part of an ecological system.

ecological process: something which changes the value of one or more ecological attributes of one or more [ecological] entities.

ecological schema: An equation for computing some ecological quantity and the ecological context in which it applies.

ecological system level: consists of information describing a specific (real or hypothetical) ecological system to be modelled (may be generalised to "actual system level" for modelling other domains.)

ecological value: a value of an ecological attribute

general/ecological level: consists of knowledge that is accepted to be universally true. (may be generalised to "general/domain level" for modelling other domains.)

effect variable: variables that some effect of some process gives rise to.

entity type: a special class of types which includes the most general type of entity (*entity*) and all its subtypes but excludes every other type (*e.g.* mappings and tuples).

entity type function: a type function that returns an entity type
e.g. *set*, v

exogenous variable: model variable that changes over time, but does not depend on any model variable

expressive power: the ability of a formalism to represent and reason about the required information and distinctions in a domain.

formalism: a set of syntactic and semantic conventions for describing something. (synonym for 'language')

fringe benefit: a benefit which was more a result of other design decisions, rather than itself being a design requirement. (*e.g.* space efficiency)

function qualifier: second order function which may be used to induce new functions from existing functions. (*e.g. average*, *rate*).

high-level goal: a goal that does not mention specific concepts about the ecological system or model (*e.g.* to enhance understanding)

idealisation: a modelling decision which entails a simplifying assumption. This assumption is embodied in the simulation model, but it does not hold in the actual system being modelled.

induced attribute: an attribute derived from existing attributes in conjunction with function qualifiers. They are inherited by either the same type of entity that the original attribute applied to or to the type of sets of that type. *e.g.* $qnam\_ent(average, weight)$

instantiated construct: an instantiation of a single construct of a formalism.

intermediate variable: model variable which depends on (*i.e.* is computed from) any other model variable, but are not state, rate variables (partial or net). They will 'happen to' be constant if they neither depend on state nor exogenous variables.

interpreter level: Information whose primary and/or sole purpose is to direct the interpreter (*i.e.* computer assistant). It relevant during the process of construction a model but not a part of the model itself. This includes goals and 'interest' specifications.

kind: *not* a technical term; to avoid mixing technical and non-technical usage of the same words we use 'kind' where 'sort' or 'type' would otherwise be appropriate.

language: a set of syntactic and semantic conventions for describing something. (synonym for 'formalism')

logical variable: a variable in a logical expression. It may be free or bound.

low-level goal: a goal that mentions specific concepts about the ecological system or model (*e.g.* affect of temperature on jellyfish)

modelling assumption: see idealisation

model parameter: something which for different runs of a model may have a different value, but is constant for a single run

model comprehension: a model can be examined and understood because an account of the model in domain terms is available. The important facts about the system being modelled are available as well as the important modelling decisions related to these facts.

net rate variable: model variable which is equal to the total rate of change of some state variable.

parameter: see model parameter

partial rate variable: model variable which is equal to the partial rate of change of the value of some state variable (usually modelling a specific effect of some process.)

partial specification: part of a specification which is presumed to consist of more than one instantiated construct.

proper constant: a quantity which is by definition constant and whose value is usually fixed by natural laws.

property: non-ecological attribute (*e.g.* orderedness)

property value: value of a property, (*i.e.* a non-ecological value)

pure model variable: a model variable that is not formally associated with its ecological meaning. The only documentation for these variables is in the form of user supplied text.

pure schema: an equation with inputs and outputs, but no explicit ecological meaning.

qualified attribute: see induced attribute

runnable model level: information that is used to represent the simulation model; a sub-level of the simulation modelling level distinguished from dialogue level.

simulation model: a computer model which is a program that simulates a dynamic system that the embodied conceptual models collectively represent

set type: a type defined using the *set* construct.

simulation modelling level: information concerned with the process of constructing a runnable model (including the runnable model itself).

sort: a primitive entity type (*i.e.* not a set type or collection type)

specification: a set of instantiated constructs.
e.g. $part(branch, tree)$

state variable: model variable whose values are computed over each time step in the simulation by incrementing and/or decrementing the value from the previous iteration. The inc/decrements are determined by the partial rate variables.

static model: a computer model which represents static information; *e.g.* a taxonomy.

syntactic adequacy: degree to which syntax of a formalism is convenient to work with.

type: that which every expression in a formal language must have. *e.g.* the type of shp is *sheep*, the type of biomass is $ecol\_ent \times time \mapsto positive$, the type of $biomass(shp, now)$ is *positive*.

type function: a function that returns a value that is a type rather than an object-level entity, relation, or function.
e.g. *qnam*

value space: the set of all possible values that an ecological attribute may have. It is a property of an ecological attribute.

Appendix B

System Dynamics Modelling

B.1 Overview

A system dynamics model represents a system as a set of compartments with material flowing into and out of them. One can think of each compartment as a *tank*. Each tank has some filling pipes, and some emptying pipes which connect to other tanks. Each connecting pipe has a valve which governs the flow. Every system dynamics model has one special tank called the *source/sink* which is the 'outside world' with respect to the system being considered. Running a model consists of calculating the changes in the amount of material in each compartment, given some set of initial conditions and mathematical relationships governing the flows. Schematically, a system dynamics model is represented as a directed graph with the compartments as nodes, and flows as arcs.

Consider the following example (figure B–1): we wish to model sheep grazing in a particular area. We represent the sheep and grass in terms of their calorific (energy) content. There are thus two compartments: SHEEP and GRASS. The material that flows is ENERGY. There are four significant processes represented in this model: photosynthesis, grazing, sheep respiration, and grass respiration[1]. Each is represented by a flow. The direction is indicated by the heavy arrows in the figure. Note that all of the flows except grazing involve the source/sink which is not explicitly represented in the figure.

Each compartment has an associated *state variable* whose value represents the contents of the compartment. Each flow has a corresponding *partial rate variable* that represents the rate of material transfer from one compartment to another due to that flow.

Mathematically, a system dynamics model may be represented by a set of differential equations. Each compartment has an associated *state variable* whose value represents the contents of the compartment. Each flow has a corresponding *partial rate variable* that represents the partial rate of material transfer from one compartment to another. We use the term 'partial' to indicate that it is not

[1] To respire here is to metabolise, *not* breathing.

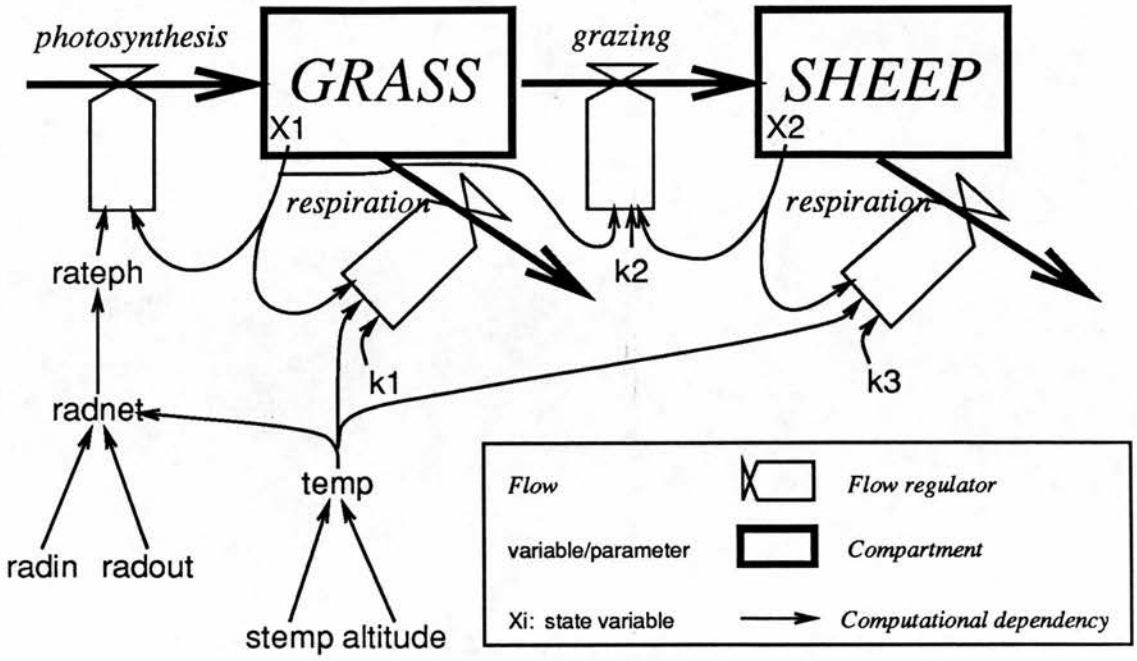


Figure B–1: Example of a System Dynamics Model

the net rate of change of the contents of a compartment due to all flows. We associate each compartment with a unique integer. Compartment number 0 is the source/sink. The contents of the ith compartment are represented by the state variable X_i. The rate of flow of material from compartment i to compartment j at time t is represented by $flow_{ij}(t)$. Although not strictly necessary, for simplicity, we assume that there is only one flow from one compartment to another. If there is no flow or if the flow goes the other way, the value is simply 0. There is one differential equation for each compartment. The left hand side of each equation represents the net rate of change of the amount of material in a compartment.

$$\frac{dX_i}{dt} = \sum_{j=0}^{n} flow_{ji}(t) - \sum_{j=0}^{n} flow_{ij}(t)$$

The rate of flow at any time depends on any number of factors. It may be constant, or may depend on other state variables. The flow from one compartment to another often depends on the current contents of either or both of the donor and recipient compartments. For instance, the rate of energy production by photosynthesis is proportional to the amount of grass energy present. The coefficient of proportionality in this case is the specific rate of photosynthesis for grass. This coefficient may be constant or it may depend on other factors (*e.g.* radiation, temperature). The specification of how to compute all the $flow_{ij}$ constitutes an acyclic computation network. The root nodes are the partial rate variables, the leaves are either constants, or state variables. The circular dependency with respect to state variables is only apparent. To be run, the differential equations are turned into difference equations and the value of the previous iteration for a state variable is used to compute its value for the current interation.

B.2 Stella

[Lewis, 1986] is a convenient tool for specifying system dynamics models. It is a commercial product with a slick graphics interface. A user directly creates and manipulates boxes and arrows corresponding to compartments and flows. A equations are specified with a separate equation editing facility. Models may be run, and the output displayed in graphical and/or tabular format.

Appendix C

Summary of ElkLogic

C.1 Object Level Constructs

Figure C–1 shows the basic syntactic constructs for ElkLogic. Figure C–2 shows the important higher-order functions in Elklogic.

C.2 Meta-Level / Implementation Constructs

Figure C–3 shows the meta-level types used. There are two kinds, the ones on top all have object-level interpretations; the others do not. Figure C–4 shows the key meta-level constructs for creating the object level functions and relations that represent ecological attributes, effecte, and model variables.

	Arithmetic
$+, -,$	addition, subtraction
$\cdot, /$	multiplication, division
	Logical Connectives, Quantifiers, etc.
$=$	equality
$\neg$	negation
$\wedge$	conjunction
$\vee$	disjunction
$\rightarrow$	implication
$\leftrightarrow$	equivalence
$\forall$	universal quantification
$\exists$	existential quantification
λ	lambda abstraction
	Miscelleneous
$\{\}$	for sets
$()$	for tuples and functions
	Type Constructors
$\mathcal{S}$	set of sorts
set	set formation
$set^{(i)}$	nested *set*; $set(set(\ldots))$
$\sqcup$	type union
$\setminus_t$	type complement
$\#$	collection; defined in terms of *set*, $\sqcup$, and $\setminus_t$
$\odot$	collection or individual
$\mathcal{P}$	power type
$\times$	tuple formation
$\mapsto$	function mapping
T^n	shorthand for $T \times T \ldots \times T$ for n occurrences of the type T.
	Relations
$:_\circ$	basic instance
$:$	instance
$\in$	set membership
$\cup$	set union
$\sqsubset^s_\circ$	basic subsort relation; defines the sort hierarchy
$\sqsubset^s$	proper subsort relation; transitive closure of $\sqsubset^s_\circ$
$\sqsubseteq^s$	subsort relation; reflexive version of $\sqsubset^s$
$\sqsubset$	proper subtype relation; induced from $\sqsubset^s_\circ$, *set*, $\sqcup$, and $\setminus_t$
$\sqsubseteq$	subtype; reflexive version of $\sqsubset$
$\prec^p_\circ$	basic possible part
$\prec^p$	possible part; transitive closure of $\prec^p_\circ$
$\subset^p_\circ$	basic possible component; induced from $\sqsubset^s_\circ$, $\#$, and $\prec^p_\circ$
$\subset^p$	possible component; transitive closure of $\subset^p_\circ$
$\subset_\circ$	basic component
$\subset$	proper component; transitive closure of $\subset_\circ$
$\subseteq$	component; reflexive version of $\subset$

Figure C–1: ElkLogic: Fundamentals

Higher Order Functions	
rate *average* *total* *maximum* *minimum*	Maps unary functions to unary functions Given a unary function and a set, returns the average, (total, minimum, maximum) value of the function applied to each member of the set.
qnam_tim	Induces new attributes from existing ones and one of *average*, *maximum*, etc. The new attribute applies over sets of times.
qnam_ent	Induces new attributes from existing ones and one of *average*, *maximum*, etc. The new attribute applies to sets of (non-time) entities.
qnam	Induces new model variables from existing ones and one of *average*, *maximum*, etc. The new variable applies over sets of times.
v	value type inducer
attvar_fn	represents the ecological meaning of proper model variables
attparm_fn	represents the ecological meaning of model parameters
effvar_fn	represents the ecological meaning of effect variables
Indexing	
$tim\_dim\_fn_n$	for representing time substructure
$att\_dim\_fn_n$	for representing attribute-based substructure

Figure C–2: Higher Order Functions

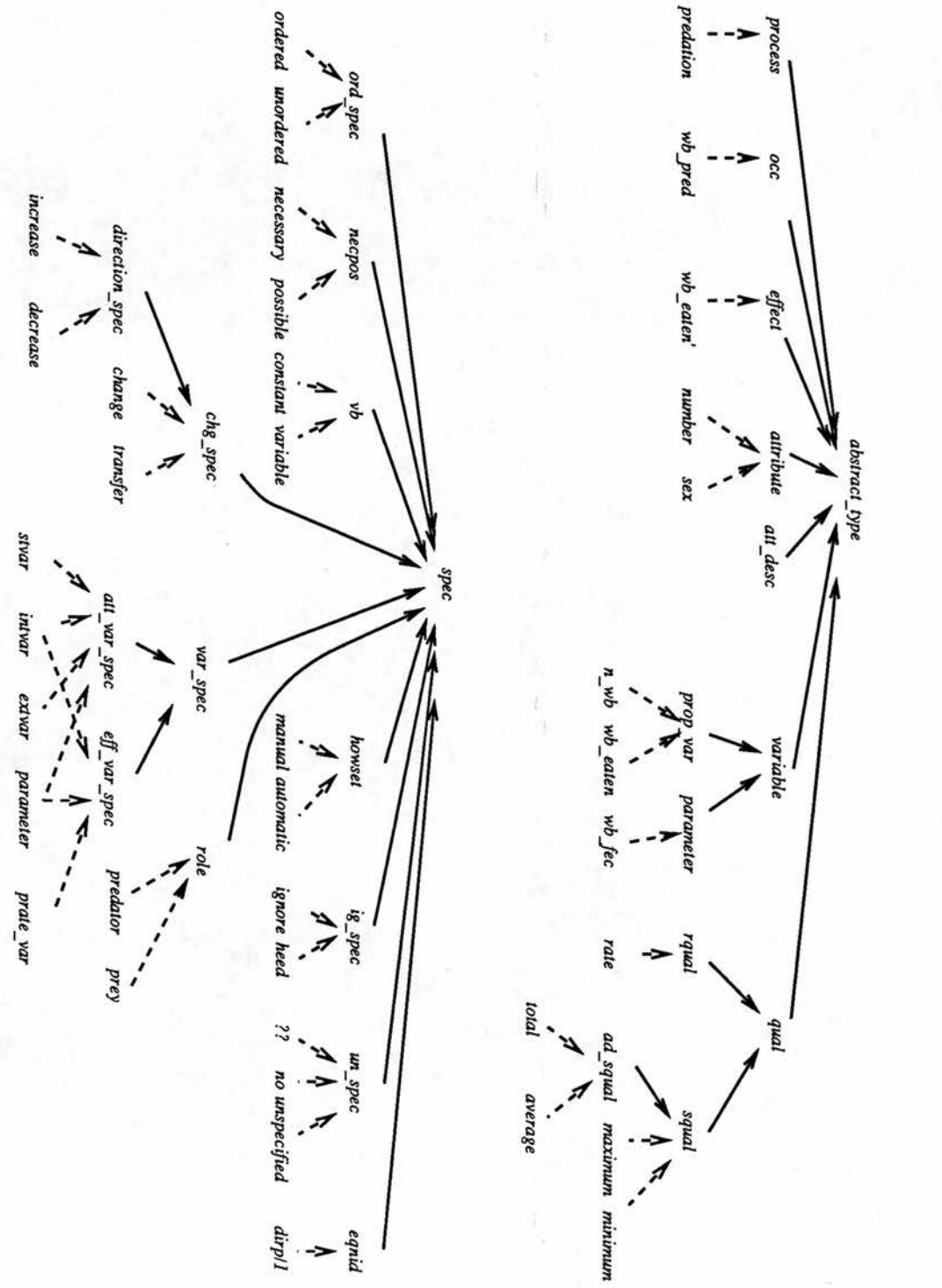


Figure C–3: Meta-Level Types

Attributes and Values	
qualitative	$attribute \times set(value) \mapsto bool$
att_def	$attribute \times \mathcal{P}(ecol\_ent) \times (\#value)^2 \times (vb \sqcup time) \times value \mapsto bool$
att_inh	$attribute \times \mathcal{P}(ecol\_ent) \times (\#value)^2 \times var\_spec \times value \times howset \mapsto bool$
att_var	$attribute \times ecol\_ent \times howset \times variable \times \#value \times var\_spec$
Processes	
role_def	$process \times ecol\_ent \times agency \mapsto bool$
effect_def	$process \times attribute \times role^{\odot} \times chg\_spec \times time \times necpos \mapsto bool$
role_esys	$occ \times role \times ecol\_ent \mapsto bool$
effect_esys	$occ \times attribute \times role \times time \times chg\_spec \mapsto bool$
effect_inh	$occ \times attribute \times role \times chg\_spec \times time \times howset \mapsto bool$
effect_var	$occ \times attribute \times role^{\odot} \times howset \times variable \times chg\_spec \times time \times eqnid \mapsto bool$
Model Variables	
pure_model_var	$variable \times ord\_spec \times \#value \times var\_spec \times value \sqcup eqnid \times text \mapsto bool$
pure_prate_var	$variable \times ord\_spec \times \#value \times time \times value \sqcup eqnid \times variable^2 \times text$
state_variable	$variable \times \#value \times value \times text \mapsto bool$
parameter_variable :	$variable \times \#value \times value \times text \mapsto bool$
exogenous_variable :	$variable \times \#value \times eqnid \times text \mapsto bool$
intermediate_variable :	$variable \times \#value \times eqnid \times text \mapsto bool$
prate_variable :	$variable \times \#value \times eqnid \times variable^2 \times text \mapsto bool$

Figure C–4: Attributes, Values, Procesese, Variables

Appendix D

Serengeti Formalised

D.1 Kernal

```
% File:     systemkb.pl
% Author:   Mike Uschold
% Updated:  Sunday Jul 30  3:44   (1989)
% Purpose:  Specifies all system related components of the knowledge base
%           which are potentially useful for *every* application.
%           What goes in here is to some extent a matter of choice.  It
%           could vary somewhat for very different classes of users.
%           This includes:
%                1. Sorts
%                2. Isa Relationships
%                3. Instances (usually implied)
%                4. Component Relationships (usually implied)
%                5. Functions (e.g. "number_of_members" for set sorts)

% PREDICATE
% System Sorts
% system_base_sort(S)

system_base_sort(indiv).
system_base_sort(ecol_indiv).
system_base_sort(time).
system_base_sort(value).
system_base_sort(real).
system_base_sort(positive).
system_base_sort(natural).
system_base_sort(integer).
system_base_sort(v(A)) :-
    \+ var(A),
    att_def(A, _).
system_base_sort(Time) :-
   time_measure(_, Time, _).

% A simple collection sort of a base sort is a valid sort


% Predicate
collection_type(#S) :-
    base_sort(S).

% PREDICATE
% System Isa Relationships
% Simple case:
system_base_isa(ecol_entity,indiv).
system_base_isa(ecosystem, ecol_entity).
```

```
system_base_isa(time,indiv).
system_base_isa(value,indiv).
system_base_isa(real,value).
system_base_isa(natural,positive).
system_base_isa(natural,integer).
system_base_isa(integer,real).
system_base_isa(positive,real).
system_base_isa(v(A), value) :-
    \+ var(A),
    att_def(A, _).
system_base_isa(Time, time) :-
    time_measure(_, Time, _).

% PREDICATE
% System Instances
% system_inst(O, S)
% eg system_inst(typical(wb_pop), wb) if inst(wb_pop,#wb)
% system_inst(typical(O),S) :-
%     inst(O,#S).


% Numeric Instances
system_inst(N, natural) :-
    integer(N),
    N >= 0, !.
system_inst(N, integer) :-
    integer(N), !.
system_inst(P, positive) :-
    float(P),
    P >= 0, !.
system_inst(P, real) :-
    float(P).

% Value Space Instances
system_inst(V, T) :-
    qualitative_inst(V, T).

% Category Instances
system_inst(C, #S) :-
    explicit_category(C ,S).

% PREDICATE
qualitative_inst(V, v(A)) :-
    qualitative(A, Values),
    member(V, Values).

% PREDICATE
% System Attributes
system_att_def(number, #indiv,
               att_def_desc(ordered,integer,na,variable,na)).


% Qualitative Value Spaces
system_qualitative(sex,[male,female]).

% Inheritance does not work for this as for other attributes.
system_proc_att_def(spr_pred(Predator), SPrey,
               att_def_desc(ordered,['positive'],na,variable,na)) :-
    participates(predation, [predator:Predator,prey:Prey]),
    inst(Prey, SPrey).
```

D.2 Serengeti Specification

```
base_att_obj_interest(weight,grs,user,heed).
base_att_obj_interest(area,plains,user,ignore).

user_base_sort(phys_obj).
user_base_sort(lifeform).
user_base_sort(plant).
user_base_sort(animal).
user_base_sort(place).
user_base_sort(region).
user_base_sort(wb).
user_base_sort(lifeform).
user_base_sort(plant).
user_base_sort(grass).


user_base_isa(phys_obj, ecol_entity).
user_base_isa(lifeform,phys_obj).
user_base_isa(plant,lifeform).
user_base_isa(animal,lifeform).
user_base_isa(place, ecol_entity).
user_base_isa(region, place).
user_base_isa(wb,animal).
user_base_isa(grass,plant).

user_base_part(region,region).

user_base_att_def(sex,lifeform,att_def_desc(unordered,[male,female,neuter],[male,female,neuter],constant,na))
user_base_att_def(r_surv,animal,att_def_desc(ordered,real,na,constant,na)).
user_base_att_def(fecundity,animal,att_def_desc(ordered,positive,na,constant,na)).
user_base_att_def(weight,phys_obj,att_def_desc(ordered,positive,na, constant,na)).
user_base_att_def(age,phys_obj,att_def_desc(ordered,positive,na,constant,na)).
user_base_att_def(biomass,lifeform,att_def_desc(ordered,real,na,variable,na)).
user_base_att_def(area, place, att_def_desc(ordered,positive,na,variable,na)).
user_base_att_def(htime,animal,att_def_desc(ordered,positive,na,constant,na)).
user_base_att_def(cap_cf,animal,att_def_desc(ordered,real,na,constant,na)).
user_base_att_def(amount_water,region,att_def_desc(ordered,real,na,variable,na)).
user_base_att_def(amount_ddt,ecol_indiv,att_def_desc(ordered,real,na,variable,na)).
user_base_att_def(pop_density,#animal,att_def_desc(ordered,positive,na,variable,na)).
user_base_att_def(rainfall,place,att_def_desc(ordered,positive,na,variable,na)).

base_att_inh(htime,animal,att_def_desc(ordered,positive,na,parameter,1,possible)).


base_att_use(weight,grs,user,att_use_desc(grs_wt,ordered,positive,intvar,eqn9)).
base_att_use(number,wb_pop,user,att_use_desc(n_wb,ordered,integer,stvar,10000)).
base_att_use(pop_density,wb_pop,user,att_use_desc(wb_density,ordered,positive,intvar,eqn5)).
base_att_use(qnam_ent(average,r_surv),wb_pop,manual,att_use_desc(spr_wb_surv,ordered,real,parameter,0.95)).
base_att_use(qnam_ent(average,htime),wb_pop,manual,att_use_desc(wb_htime,ordered,positive,parameter,0.0799999
base_att_use(qnam_ent(average,cap_cf),wb_pop,manual,att_use_desc(wb_cap_cf,ordered,real,parameter,317)).
base_att_use(qnam_ent(average,fecundity),wb_pop,manual,att_use_desc(wb_fec,ordered,real,parameter,0.5)).
base_att_use(qnam_ent(average,cap_cf),alt_prey,manual,att_use_desc(ap_cap_cf,ordered,real,parameter,100)).
base_att_use(qnam_ent(average,htime),alt_prey,manual,att_use_desc(ap_htime,ordered,positive,parameter,0.05)).
base_att_use(rainfall,serengeti,manual,att_use_desc(dry_ssn_rain,ordered,positive,parameter,250)).
base_att_use(area,serengeti,manual,att_use_desc(area_sgti,ordered,positive,parameter,1000000)).
base_att_use(qnam_ent(average,r_surv),wb_calves,manual,att_use_desc(spr_cf_surv,ordered,real,intvar,??)).
base_att_use(pop_density,alt_prey,manual,att_use_desc(ap_density,ordered,positive,intvar,??)).
base_att_use(spr_pred(predators),wb_pop,manual,att_use_desc(spr_pred_wb,ordered,positive,intvar,??)).
base_att_use(number,predators,manual,att_use_desc(n_pred,ordered,integer,parameter,1200)).
base_att_use(number,alt_prey,manual,att_use_desc(n_aprey,ordered,integer,parameter,4200)).
base_att_use(area,plains,manual,att_use_desc(area_plains,ordered,positive,parameter,20000)).

pure_model_variable(cf_born,pure_var_desc(ordered,positive,intvar,eqn3,'No wb calves born per year')).

user_base_inst(wb_pop,#wb).
```

```
user_base_inst(alt_prey,#animal).
user_base_inst(predators,#animal).
user_base_inst(grs,grass).
user_base_inst(nw_corridor,region).
user_base_inst(woodland,region).
user_base_inst(wb_calves,#wb).
user_base_inst(serengeti,region).
user_base_inst(plains,region).
user_base_inst(n_woodland,region).
user_base_inst(cent_woodland,region).


base_inst_component(wb_calves,wb_pop,description(#wb,#wb,subdiv,subdiv)).
base_inst_component(cent_woodland,woodland,description(region,region,trans(member: #region,coll_part),part)).
base_inst_component(n_woodland,woodland,description(region,region,trans(member: #region,coll_part),part)).
base_inst_component(woodland,serengeti,description(region,region,trans(member: #region,coll_part),part)).
base_inst_component(plains,serengeti,description(region,region,trans(member: #region,coll_part),part)).
base_inst_component(nw_corridor,serengeti,description(region,region,trans(member: #region,coll_part),part)).

user_role_def(predation,predator,#animal,active).
user_role_def(predation,prey,#animal,passive).
user_role_def(predation,'','',other).
user_role_def(precipitation,agent,region,active).
user_role_def(precipitation,'','',passive).
user_role_def(precipitation,'','',other).
user_role_def(grazing,grazer,animal,active).
user_role_def(grazing,grazed,plant,passive).
user_role_def(grazing,'','',other).
user_role_def(mortality,agent,#lifeform,active).
user_role_def(mortality,'','',passive).
user_role_def(mortality,'','',other).
user_role_def(reproduction,agent,#lifeform,active).
user_role_def(reproduction,'','',passive).
user_role_def(reproduction,'','',other).

affected_atts_def(predation,[number],[amount_ddt]).
affected_atts_def(precipitation,[amount_water],[amount_ddt]).
affected_atts_def(grazing,[biomass,weight],[amount_ddt]).
affected_atts_def(mortality,[number],[]).
affected_atts_def(reproduction,[number],[]).
affected_atts_esys(wb_pred,predation,[number],[]).
affected_atts_esys(wd_precip,precipitation,[amount_water],[amount_ddt]).
affected_atts_esys(wb_mort,mortality,[number],[]).
affected_atts_esys(wb_rep,reproduction,[number],[]).
affected_atts_esys(ap_pred,predation,[number],[]).


user_effect_def(predation,number,prey,effect_def_desc(decrease,month,necessary)).
user_effect_def(precipitation,amount_water,agent,effect_def_desc(increase,fortnight,necessary)).
user_effect_def(precipitation,amount_ddt,agent,effect_def_desc(increase,month,possible)).
user_effect_def(grazing,biomass,grazed,effect_def_desc(decrease,hour,necessary)).
user_effect_def(grazing,biomass,grazer,effect_def_desc(increase,hour,necessary)).
user_effect_def(grazing,weight,grazed,effect_def_desc(decrease,hour,necessary)).
user_effect_def(grazing,weight,grazer,effect_def_desc(increase,hour,necessary)).
user_effect_def(grazing,amount_ddt,grazer,effect_def_desc(increase,week,possible)).
user_effect_def(grazing,amount_ddt,grazed,effect_def_desc(decrease,week,possible)).
user_effect_def(mortality,number,agent,effect_def_desc(decrease,day,necessary)).
user_effect_def(reproduction,number,agent,effect_def_desc(increase,second,necessary)).

user_role_esys(wb_pred,predator,predators).
user_role_esys(wb_pred,prey,wb_pop).
user_role_esys(wd_precip,agent,woodland).
user_role_esys(wb_mort,agent,wb_pop).
user_role_esys(wb_rep,agent,wb_pop).
user_role_esys(ap_pred,predator,predators).
user_role_esys(ap_pred,prey,alt_prey).
```

```
user_effect_use(wb_mort,number,agent,effect_use_desc(wb_die,decrease,year,'')).
user_effect_use(wb_pred,number,prey,effect_use_desc(wb_eaten,decrease,year,'')).
user_effect_use(wb_rep,number,agent,effect_use_desc(wb_repro,increase,year,'')).



inarc('a.1','dirp/1',a).
inarc('x.1','dirp/1',x).
inarc('a.2','dirp/2',a).
inarc('x.2','dirp/2',x).
inarc('food.1','food_surv/1',food).
inarc('rainfall.1','rain_growth/1',rainfall).
inarc('x.4','dircomp/1',x).
inarc('a.3','dircomp/1',a).
inarc('x.5','dirp/3',x).
inarc('a.4','dirp/3',a).
inarc('number($pop,t).2','pop_density/9','number($pop,t)').
inarc('area($region,t).2','pop_density/9','area($region,t)').
inarc('pop_density(wb_pop,t).0','spr_pred/1','pop_density(wb_pop,t)').
inarc('pop_density(alt_prey,t).0','spr_pred/1','pop_density(alt_prey,t)').
inarc('qnam_ent(average,htime)app (wb_pop,t).1','spr_pred/2','qnam_ent(average,htime)app (wb_pop,t)').
inarc('qnam_ent(average,cap_cf)app (wb_pop,t).1','spr_pred/2','qnam_ent(average,cap_cf)app (wb_pop,t)').
inarc('pop_density(wb_pop,t).1','spr_pred/2','pop_density(wb_pop,t)').
inarc('qnam_ent(average,htime)app (alt_prey,t).1','spr_pred/2','qnam_ent(average,htime)app (alt_prey,t)').
inarc('qnam_ent(average,cap_cf)app (alt_prey,t).1','spr_pred/2','qnam_ent(average,cap_cf)app (alt_prey,t)').
inarc('pop_density(alt_prey,t).1','spr_pred/2','pop_density(alt_prey,t)').
inarc('number($pop,t).3','pop_density/10','number($pop,t)').
inarc('area($region,t).3','pop_density/10','area($region,t)').

outarc(ap_density,'pop_density/10','pop_density($pop,t)').
outarc(spr_pred_wb,'spr_pred/2','spr_pred($predator)app ($prey,t)').
outarc(wb_density,'pop_density/9','pop_density($pop,t)').
outarc(wb_eaten,'dirp/3',f).
outarc(wb_die,'dircomp/1',f).
outarc(cf_born,'dirp/1',f).
outarc(wb_repro,'dirp/2',f).
outarc(spr_cf_surv,'food_surv/1',surv).
outarc(grs_wt,'rain_growth/1',amount).


tie('area($region,t).3',area_sgti).
tie('number($pop,t).3',n_aprey).
tie('pop_density(alt_prey,t).1',ap_density).
tie('qnam_ent(average,cap_cf)app (alt_prey,t).1',ap_cap_cf).
tie('qnam_ent(average,htime)app (alt_prey,t).1',ap_htime).
tie('pop_density(wb_pop,t).1',wb_density).
tie('qnam_ent(average,cap_cf)app (wb_pop,t).1',wb_cap_cf).
tie('qnam_ent(average,htime)app (wb_pop,t).1',wb_htime).
tie('a.4',spr_pred_wb).
tie('pop_density(alt_prey,t).0',ap_density).
tie('pop_density(wb_pop,t).0',wb_density).
tie('area($region,t).2',area_sgti).
tie('number($pop,t).2',n_wb).
tie('x.5',n_pred).
tie('a.3',spr_wb_surv).
tie('x.4',n_wb).
tie('x.1',n_wb).
tie('a.1',wb_fec).
tie('a.2',spr_cf_surv).
tie('x.2',cf_born).
tie('food.1',grs_wt).
tie('rainfall.1',dry_ssn_rain).


imod('dirp/2',dirp,[x,a]).
imod('dirp/1',dirp,[x,a]).
imod('rain_growth/1',rain_growth,[rainfall]).
```

```
imod('food_surv/1',food_surv,[food]).
imod('dircomp/1',dircomp,[x,a]).
imod('dirp/3',dirp,[x,a]).
imod('pop_density/9',pop_density,['number($pop,t)','area($region,t)']).
imod('spr_pred/2',spr_pred,['qnam_ent(average,htime)app (wb_pop,t)','
                            qnam_ent(average,cap_cf)app (wb_pop,t)','
                            pop_density(wb_pop,t)','
                            qnam_ent(average,htime)app (alt_prey,t)','
                            qnam_ent(average,cap_cf)app (alt_prey,t)','
                            pop_density(alt_prey,t)']).
imod('pop_density/10',pop_density,['number($pop,t)','area($region,t)']).

var_inputs_equation('spr_pred/2',wb_cap_cf*wb_density/
         (1+ (wb_htime*wb_cap_cf*wb_density+ap_htime*ap_cap_cf*ap_density))).

output_variable(n_wb).
output_variable(spr_pred_wb).


runnable_model_goals(
   (eval_n_assign(rain_growth,[dry_ssn_rain],grs_wt)','
    eval_n_assign(pop_density,[n_aprey,area_sgti],ap_density)','
    eval_n_assign(food_surv,[grs_wt],spr_cf_surv)),

  ((eval_n_assign(dircomp,[n_wb,spr_wb_surv],wb_die)','
    eval_n_assign(pop_density,[n_wb,area_sgti],wb_density)','
    eval_n_assign(dirp,[wb_fec,n_wb],cf_born)','
    eval_n_assign(spr_pred,[wb_htime,wb_cap_cf,wb_density,
                            ap_htime,ap_cap_cf,ap_density],spr_pred_wb)','
    eval_n_assign(spr_pred,[wb_htime,wb_cap_cf,wb_density,
                            ap_htime,ap_cap_cf,ap_density],spr_pred_wb)','
    eval_n_assign(dirp,[n_pred,spr_pred_wb],wb_eaten)','
    eval_n_assign(dirp,[spr_cf_surv,cf_born],wb_repro))','

    eval_n_assign(stvar,[n_wb,[wb_repro],[wb_die,wb_eaten]],n_wb)).
```